Proceedings of the

2005 Automated Software Engineering Workshop on Software Certificate Management (SoftCeMent'05)

Long Beach, California, USA
November 8, 2005

Sponsored by the

Association for Computing Machinery

Special Interest Group on Software Engineering (SIGSOFT)

Special Interest Group on Artificial Intelligence (SIGART)

Foreword

This volume contains the position papers presented at the Workshop on Software Certificate Management (SoftCeMent'05), held in conjunction with the 20th IEEE/ACM International Conference on Automated Software Engineering, on November 8th, 2005, in Long Beach, California.

Software certification demonstrates the reliability, safety, or security of software systems in such a way that it can be checked by an independent authority with minimal trust in the techniques and tools used in the certification process itself. It can build on existing validation and verification (V&V) techniques but introduces the notion of explicit software certificates, which contain all the information necessary for an independent assessment of the demonstrated properties. Software certificates support a product-oriented assurance approach, combining different techniques and forms of evidence (e.g., fault trees, "sign-offs", safety cases, formal proofs, ...) and linking them to the details of the underlying software.

A software certificate management system provides the infrastructure to create, maintain, and analyze software certificates. It combines functionalities of a database (e.g., storing and retrieving certificates) and a make-tool (e.g., incremental re-certification). It can also maintain links between system artifacts (e.g., design documents, engineering data sets, or programs) and different varieties of certificates, check the validity of certificates, provide access to explicit audit trails, enable browsing of certification histories, and enforce system-wide certification and release policies. It can at any time provide current information about the certification status of each component in the system, check whether certificates have been audited, compute which certificates remain valid after a system modification, or even automatically start an incremental recertification.

The main goal of this workshop is to explore new technologies, underlying principles, and general methodologies for supporting software certificate management.

Ewen Denney **Bernd Fischer** **Mark Jones** **Dieter Hutter**

Organizers

Program Committee

Table of Contents

Schedule

All talks will be held at the Renaissance Ballroom 3.

Breakfast

8:00-8:30 Main lobby

Opening

8:45-9:00 Welcome

Session 1:

9:00-9:30 Ewen Denney, Bernd Fischer:
Software Certification and Software Certificate Management Systems

9:30-10:00 Mark P. Jones:
Evidence Management in Programatica

10:00-10:30 Andrew Kornecki, Janusz Zalewski:
Qualification of Software Development Tools for Airborne Systems Certification

Coffee

10:30-11:00 Main lobby

Session 2:

11:00-11:30 Mark Sherriff, Laurie Williams:
A Method for Verification and Validation Certificate Management in Eclipse

11:30-12:00 Mike Whalen:
Certificate Management: A Practitioner's Perspective

Lunch

12:00-14:00 Lunch at the Rockbottom Brewery

Session 3:

14:00-14:30 Discussion

14:30-15:00 Graeme Parkin, Peter Harris:
Certifying Software Fit For Purpose

15:00-15:30 Andrew Ireland:
On the Scalability of Proof Carrying Code for Software Certification

Coffee

15:30-16:00 Main lobby

Session 4:

16:00-16:30 Songtao Xia, Ben Di Vito:
Software Certification for Temporal Properties with Affordable Tool Qualification

16:30-17:00 Bernhard Beckert, Thorsten Bormer, Vladimir Klebanov:
Reusing Proofs when Program Verification Systems are Modified

17:00-17:30 Dieter Hutter:
Software Certification Management: How Can Formal Methods Help?

Closing Remarks

Software Certification and Software Certificate Management Systems

(Position Paper)

Ewen Denney and Bernd Fischer

USRA/RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA
{edenney,fisch}@email.arc.nasa.gov

1 Introduction

Software certification demonstrates the reliability and safety of software systems in such a way that it can be checked by an independent authority with minimal trust in the techniques and tools used in the certification process itself. It builds on existing software assurance, validation, and verification techniques but introduces the notion of *explicit software certificates*, which contain all the information necessary for an independent assessment of the demonstrated properties. A *software certificate management system* (SCMS) provides a range of certification services. It maintains the links between different system artifacts (e.g., design documents, engineering data sets, or programs) and different varieties of certificates, checks the validity of certificates, provides access to explicit audit trails, enables browsing of certification histories, and enforces system-wide certification and release policies.

We believe that a customizable SCMS with support for automated re-certification of diverse artifacts should become an essential part of any effective development process. Its primary impact is to increase the reliability and safety of software systems by providing automation support for their audit. A SCMS can at any time provide current information about the certification status of each component in the system, check whether certificates have been audited, compute which certificates remain valid after a system modification, and automatically start an incremental re-certification.

We are particularly interested in the combination of software certification with automated code generation and formal verification methods. Here, our focus is on the related questions of how code generators can support the certification process, and how software certification can be used to demonstrate and increase the reliability of the code generation process.

2 Challenges

Building reliable software is a challenging task in itself, but there are several challenges specifically related to certification, e.g.,

– maintaining high reliability, especially when a combination of diverse development techniques is used,

– minimizing certification efforts, especially for product families and interconnected systems of systems,
– reducing certification and re-certification times,
– linking between artifacts and certificates, and
– providing useful information (e.g., estimates of certification efforts).

We claim that the solution to these challenges is an intelligent, automated, and highly customizable software certificate management system integrated into the development process.

3 Software Certification

Software certification comprises a wide range of formal, semi-formal, and informal assurance techniques, including formal verification of compliance with explicit safety policies, system simulation, testing, code reviews and human "sign offs", and even references to supporting literature. Consequently, the certificates can have different types, and the certification process requires different mechanisms. A SCMS must be able to support such different certificate types and certification mechanisms. In order to guarantee separation of concerns and thus achieve scalability, certification approaches need to concentrate on individual risk factors one at a time. Consequently, a SCMS must be able to combine different certificates for the same artifact to construct an overarching certificate and, ultimately, to provide a higher degree of confidence.

Certificates A certificate contains all information necessary for an independent assessment of the properties claimed for an artifact. Obviously, the exact nature of the certificates depends on the nature of the artifact, the property, and the claim. However, a SCMS needs a unified view of certificates. At its most abstract, a certificate thus has to represent the three entities involved in the certification process, (i) the artifact being certified, (ii) the property being asserted, and (iii) the certification authority.

Certifiable Artifacts Certifiable artifacts include not only the conventional software artifacts (e.g., product families, completed systems, individual components, or even code fragments) but also supporting non-software artifacts: requirements documents, system designs, component specifications, test plans, individual test cases, scientific and engineering data sets, and others.

In particular, the supporting evidence for one certificate can be considered as the artifact of another certificate. For example, if the correctness of a component is to be certified using traditional black-box testing, the test harness and the test scripts are supporting evidence for the certificate; at the same time, the test harness can itself be certified, e.g., by a code review, and is thus the artifact of another certificate.

Certificate Hierarchies As indicated in the example above, the certificates for an artifact are not an unstructured collection but exhibit some hierarchical structure. This structure is determined by two independent dimensions, (i) the system structure, and (ii) the certificate types.

The internal structure of a *system* is reflected in the certificate hierarchy. If a system is decomposed into a number of subsystems, and each subsystem is built from a number of components, then a certificate for the system depends directly on the certificates of

the subsystems and indirectly on the certificates of all involved components. A SCMS must be able to represent this structure, taking into account language-specific visibility rules like module and subsystem boundaries that can limit the propagation of changes.

The second dimension is given by the certificates themselves, or more precisely, by the certificate types. The validity of a certificate can also depend on certificates for the supporting evidence (as described above), or even the authority, e.g., when a code review can only be signed by a certified software engineer. This part of the certificate hierarchy reflects the internal structure and procedures of the *organization* developing the software. A SCMS can then use the certificate hierarchy for auditing and incremental re-certification, similar to the way the Unix make-tool uses explicit dependencies and rules for incremental re-compilation. The SCMS can determine which certificates need to be inspected, recomputed, or revalidated after an artifact or a certificate has been (or would be) modified.

Certifiable Properties and Certification Authorities Traditional V&V has only addressed a restricted range of formal properties. Realistically, however, software development requires a wide range of notions of software reliability, safety, and validity, each with an appropriate certification authority. This must all be supported by a customizable SCMS. Examples include coding standards, test cases, statistical validity for data sets, simulation on high-fidelity test beds, fault tree analysis (FTA), failure modes and effects analysis (FMEA), stress tests, interoperability, usability, compatibility, and feasibility studies, as well as formally specified logical safety properties.

Release Policies In the context of certification, a release refers to the transition of an artifact into a new defined state: for example, launch, system integration test, alpha and beta testing phases, spiral anchor-point milestones, or code inspection. A release policy formally describes under which conditions an artifact is deemed to be in an adequately certified state and can thus be released safely to another state. Different release policies can be formulated to describe the different types of releases, and the corresponding certification requirements.

4 Certification Services

Intuitively, a SCMS combines the functionalities of a database (e.g., storing and retrieving certificates) and a make-tool (e.g., incremental re-certification). Specifically, it provides a variety of different services.

Certificate construction The main task of the SCMS is the construction of certificates. Given an artifact, a claimed property, and a certification authority, the SCMS will attempt to construct the certificate, invoking automatic mechanisms and notifying individuals of pending tasks, as appropriate. It should estimate the time and effort that the certification will take.

Editing and revoking Users can deem an individual certification authority to no longer be valid (e.g., a bug is discovered in a test harness, or an employee's badge has expired). The SCMS should revoke all certificates which depend on this.

Certificate maintenance The SCMS will carry out intelligent re-certification when a (customizably) appropriate change has taken place in the code or, more generally,

software artifacts to be certified. Existing (sub-) certificates should be reused where possible, especially where product families are concerned.

Auditing Since the SCMS provides a complete certification history with full information about all procedures followed, comprehensive audits can be carried out, applying alternative tools and/or oversight to any elements. The audit itself can then be recorded in the certification database.

Schema management Clearly, the SCMS must be generic. It must be customizable to existing procedures. It can be thought of as having a client-server architecture. The SCMS is the client and allows users to "plug and play" with arbitrary certificate servers.

5 Current Technology and Advances Required

In terms of computing infrastructure, the notions of certificate and certification are usually used in the context of security mechanisms for computer systems, in particular for computer networks and network-based services.

A SCMS can build on an existing secure infrastructure, e.g., PKI, for distribution, authentication, tamper-proof access control, persistence, and other desirable properties. However, there are a number of differences from existing technology where advances are required:

- linking to (and deep into) software artifacts,
- the wide diversity of forms of certification, both formal and informal, and
- the need for customizability and extensibility.

The SCMS should be an integral part of a development tool suite and use the same underlying datastructures, e.g., development graphs [1]. It can be linked to other tools, e.g., code generators and software reliability estimators. In particular, software certification can be combined with automated design documentation, so that the SCMS can provide an integrated exploration tool for code, certificates, and documentation, similar to safety cases [5] but more specific to the code level. Likewise, model-based software development tools should allow the definition of arbitrary domain-specific certificate types with respect to explicit domain models.

6 Certification of Automatically Generated Code

We are currently investigating some of these ideas in the context of an ongoing project on automated code generation. We have developed an approach to safety verification [3] in which the code generator is extended to enable Hoare-style safety proofs for each individual generated program. The key idea is to generate logical annotations along with the code, so that the proofs can be automated. These proofs ensure that the generated code does not violate certain conditions during its execution. However, it has gradually become clear that since this process produces a large number of auxiliary artifacts, and involves many components of varying complexity and reliability, that additional tool support should enable users to browse the entire set of safety artifacts.

In [4], we describe a rudimentary *certification browser*, which provides linking between the generated program, its verification conditions, the generated axioms, the proofs, and the proof checks. This is a first step towards an interactive tool which would, for example, allow designated users to sign off on otherwise unverified lines of code. This would be a prototype SCMS for code generation. Similar ideas have been investigated by the Programatica project [6], though not in connection with code generation.

In recent work we have developed an approach to inferring annotations for code produced by third-party code generators. This suggests a means of certifying the code produced by COTS code generators, and circumvents the difficulties that stem from treating these tools as black boxes.

7 Conclusions

Incremental certification and re-certification of code as it is developed and modified is a prerequisite for applying modern, evolutionary development processes, which are especially relevant for NASA. For example, the Columbia Accident Investigation Board (CAIB) report [2] concluded there is "the need for improved and uniform statistical sampling, audit, and certification processes". Also, re-certification time has been a limiting factor in making changes to Space Shuttle code close to launch time. This is likely to be an even bigger problem with the rapid turnaround required in developing NASA's replacement for the Space Shuttle, the Crew Exploration Vehicle (CEV). Hence, intelligent development processes are needed which place certification at the center of development. If certification tools provide useful information, such as estimated time and effort, they are more likely to be adopted. The ultimate impact of such a tool will be reduced effort and increased reliability.

References

[1] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The Development Graph Manager MAYA (System Description). In *Proc. 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02)*. LNCS 2422, pp. 495–501, 2002.

[2] Columbia Accident Investigation Board Report, Volume 1. `http://caib.nasa.gov/`. 2003.

[3] E. Denney and B. Fischer. Formal Safety Certification of Aerospace Software. In *Proc. Infotech@Aerospace*. AIAA, 2005. Invited talk.

[4] E. Denney and B. Fischer. A Program Certification Assistant Based on Fully Automated Theorem Provers. In *Proc. International Workshop on User Interfaces for Theorem Provers, (UITP'05)*, 2005.

[5] T. Kelly and R. Weaver. The Goal Structuring Notation – a Safety Argument Notation. In *Proc. DSN Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities*, 2004.

[6] Programatica Project. `www.cse.ogi.edu/PacSoft/projects/programatica`. 2004.

Evidence Management in Programatica

Mark P. Jones

[1] OGI School of Science & Engineering at OHSU
[2] Portland State University

This paper summarizes our efforts in the Programatica project at OGI and PSU to design new kinds of tools to support the development and certification of software systems. Our approach relies on a tight integration of program source code, embedded formal properties, and associated evidence of validity. A particular goal for the toolset is to facilitate efficient and effective use of many different kinds of evidence during project development. Our current prototype targets validation of functional (security) properties of programs written in Haskell. This tool provides connections, through a language of formal properties called P-logic, to several external validation tools and supports unit testing, random testing, automated and interactive theorem proving, and signed assertions. The underlying concepts, however, are quite general and should be easily adaptable to other programming languages and development tools, and to support a wide range of both process- and artifact-oriented based validation techniques.

1 Software Development and Evidence Management

Software developers rely on a wide variety of techniques to assure themselves (and their customers) that the system they are building will function correctly:

- In the early stages of a project, experiments, models, and prototypes can be used to gain a better understanding of system requirements, and unit tests or specific test data sets may be collected to document expected behavior.
- Developers might also use tools based on static analysis or formal methods—such as model checkers or theorem provers—to obtain evidence for key properties. Tools like these can provide strong guarantees about program behavior that are particularly important in safety or security critical applications where high levels of assurance are required. However, effective use of such tools can require significant investment in both initial training and daily use, and hence they are often considered too expensive to be used in the early, more fluid stages of project development.
- Process-oriented techniques—including, for example, code inspection and design review—are an important component of current certification methodologies and can often be used at multiple stages of the development process.

There are of course, many other techniques that could be listed with the above examples, some of which are quite general while others are applicable only in specific domains. But although there are many different techniques, there is still one important common feature: each of them results in some tangible form of *evidence* about specific properties of the software system. For example, an input

and (expected) output pair can be used to document a test case; a script or tactic can be used to record the structure of a formal proof; and detailed minutes can be used to capture the results of a code review meeting.

Some of the biggest challenges for anyone tasked with certifying the behavior or properties of a software system are in managing, maintaining, and exploiting the large and diverse volumes of evidence that are created during its development. This observation motivates the development of new tools and techniques that will allow evidence to be reused, repeated, or replayed so that validity of each component in a system can be monitored automatically and incrementally without the need to reconstruct evidence from scratch at every step, or when the development is complete.

2 The Programatica Approach to Evidence Management

Our specific goals in the Programatica project are to build tools that allow users: to capture evidence and collate it with source materials; to exploit dependencies between evidence and the programs to which it refers as a means of tracking change; to automate the process of combining and reusing evidence; and, finally, to understand, manage, and guide further development and validation efforts. In addition, we recognize that evidence may come in many different forms, and that tools must be designed to address this: a key feature of our approach is the use of a general *certificate* abstraction for encapsulating, accessing, and manipulating different forms of evidence in a uniform manner.

Our approach is intended to be quite general, but it is inspired, in part, by techniques adopted in more specialized tools. The practice of "Extreme Programming" [2], for example, encourages frequent use of testing as an integral part of coding and refactoring [4] and has stimulated the development of tools that automate the testing process. These tools, however, do not attempt to deal with or incorporate other kinds of evidence. Similarly, compilation tools (such as `make` [3]) track dependencies between source code units to minimize the need for recompilation, but they do not attempt to capture other kinds of dependencies or evidence. As a final example, some systems support "external oracles" that allow users to integrate theorem proving with other validation tools such as BDDs [5] or model checking [1]. These tools, however, focus on formal validation and do not directly address evidence capture and management.

2.1 Certificates

Programatica certificates are a mechanism for encapsulating different types of evidence. The evidence itself, as well as the internal format by which it is represented, will vary from one certificate to the next. But, from the perspective of an evidence management tool, every certificate offers the same basic interface, with attributes that describe its *sequent* and *validity*, and operations that allow certificates to be *validated* and *edited*. Each of these features is described below:

– The *sequent* of a certificate formalizes the claim that the accompanying evidence is intended to support. Sequents provide the means by which disparate kinds of evidence can be brought together in a single environment. In our current system, we write sequents as judgments, $\Gamma \vdash \Gamma'$, where the *hypotheses* in Γ and the *conclusions* in Γ' are lists of formulas over a suitably chosen specification logic. The formulas in a sequent may include direct references to variables and functions that are defined in the source text. As such, a sequent also provides a starting point for tracking dependencies between certificates and the underlying code base.
– A certificate is *valid* if its sequent is consistent with the evidence it contains. For example, a certificate with sequent $\vdash A$ is valid only if it provides evidence for A. In this way, validity serves as a contract between external tools and the evidence management system.
– The actions needed to *validate* a given certificate will depend on the type of the certificate, and may, in some cases, involve significant computation. To permit a quick test of validity, each certificate includes a flag that is set only when the certificate is known to be valid. If either the certificate itself or a part of the source text that it depends on is changed, then the flag will be reset and the full test of validity can be deferred until needed.
– The actions needed to *edit* a certificate—such as modifying it so that its validity can be established—will also depend on the type of the certificate, and may, in some cases require significant user interaction.

The Programatica certificate abstraction supports compositional certification. For example, from a certificate with sequent $A \vdash B$ and another with sequent $B \vdash C$, we can derive a compound certificate with sequent $A \vdash C$. If changes to the underlying program invalidate only one of the original certificates, then we will not need to construct new evidence for the other one or for the composite. Note also that the original two certificates could be constructed using different tools; the composite can then be tagged to reflect the set of tools that were used in its construction, and this information can be used as an indication of its pedigree. In this way, certificates and sequents provide a mechanism for integrating and reasoning with different kinds of evidence.

2.2 Certificate Servers

Certificate servers (or just 'servers') play an important role as a mechanism for creating and using different types of certificate in a uniform manner. We distinguish between: *external* servers, which are used for certificates whose evidence is supplied by external tools; and *internal* servers, which use functionality that is built in to the evidence management tool, and provide a means for combining different types of evidence.

– External servers connect the evidence management system to the external tools that are used to construct and maintain evidence. As such, external servers can be understood as software plug-ins that must be installed before certificates of a particular type can be edited and validated. External

servers are responsible for translating between the languages used in source documents and sequents and the languages used by external tools. It is the responsibility of each external server to detect and report cases where translation is not possible. A second responsibility of an external server is to capture and package context from source documents so that it can be used by the external tool. We refer to this as 'theory formation' because it corresponds to assembling a theory that includes the facts and definitions that would be needed to prove a particular theorem.
- Internal servers provide built-in functionality for generating and combining evidence. This includes 'axiom' servers that can generate and validate certificates for sequents of a particular form and 'rule' servers that can be used to combine previously constructed certificates.

Servers provide an infrastructure for theorem proving with certificates in which different servers correspond to different external oracles and inference rules. One of the most tricky design choices here is to determine how much of this machinery should be built in to the evidence management tools, and how much should be delegated to an external theorem prover.

3 Challenges for Future Work

The Programatica approach to evidence management offers a new vision for high-assurance software development and certification. Our current prototypes [6] are in an early stage of development but are designed to extend current evaluation methodologies by supporting and integrating the different kinds of evidence that they require. We hope that Programatica will also provide an evolution path for introducing and applying formal methods to document and validate essential functional properties of critical software systems at high assurance levels.

There are, however, many challenges to address and evaluate with future generations of the Programatica tools, including:

- What can a tool do to help users visualize and understand the evidence they have assembled, to prioritize future validation tasks, and to identify areas in which evidence is either lacking or weak?
- How can we deal with differing levels of trust and confidence in the different kinds of evidence, servers, and models that are used?

Certainly, some aspects of confidence and trust can be quantified. For example, if one test suite includes all of the tests from another, then the first should offer at least the same degree of assurance as the second. But many other aspects are subjective and will require a flexible tool that can be tailored to policies of an organization or to the requirements of a particular certification process.

Acknowledgments

The work described in this paper has benefited significantly from the input of Programatica team members including Thomas Hallgren, James Hook, Dick Kieburtz, Rebekah Leslie, Andrew Tolmach, and Peter White.

References

1. Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A pragmatic implementation of combined model checking and theorem proving. In *Theorem Proving in Higher Order Logics (TPHOLs)*, July 1999.
2. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
3. S.I. Feldman. Make-A program for maintaining computer programs. *Software—Practice and Experience*, 9(4), 1979.
4. Martin Fowler et al. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.
5. Michael J.C. Gordon. Reachability programming in HOL98 using BDDs. In *Theorem Proving in Higher Order Logics (TPHOLs)*, August 2000.
6. The Programatica Team. Programatica tools for certifiable, auditable development of high-assurance systems in haskell. In *High Confidence Software and Systems*, Baltimore, MD, 2003.

Position Paper Draft http://ti.arc.nasa.gov/sc05/
20th IEEE/ACM International Conference on Automated Software Engineering
Software Certificate Management Workshop, SoftCeMent05
November 8, 2005
Long Beach, California, USA

Qualification of Software Development Tools for Airborne Systems Certification

Andrew J. Kornecki,
Embry Riddle Aeronautical University, Daytona Beach, FL
Janusz Zalewski
Florida Gulf Coast University, Fort Myers, FL

Introduction - Design Tools Categorization

This position paper is a result of recent research contracted by the Federal Aviation Administration. The main objective of this research was to identify the assessment criteria that allow both developers and certifying authorities to evaluate specific safety critical real-time software development tools from the system/software safety perspective. Related objectives were: to present and evaluate the state of the art in safety critical software development tools and to establish a basis for software development tool qualification guidelines. The research included literature reviews and industry surveys, creation of tool evaluation taxonomy, installation of and experiments with the selected tools and collecting and analyzing data. The focus of this research was on real-time design tools with automatic code generation features.

During the design of a real-time system, it is important to be aware that there exist two distinct classes of modern systems exposed to environmental stimuli:

- Interactive – the computer system determines the pace of operation by granting or allocating resources to clients on request when feasible (operating systems, data bases); the concerns are deadlock avoidance, fairness, data coherence.
- Reactive – the system environment determines the pace of operation while the computer system reacts to external stimuli producing outputs in timely way (process control, avionics, signal processing); the concerns are correctness and timeliness.

Consequently, design tools that assist developers in translating the requirements into source code can be categorized into two groups, selected for this study:

- function-based, block-oriented, reactive, preferred by engineers with system and control background, and
- structure-based, object-oriented, interactive, preferred by engineers with software and computer background.

Current Practice of Development Tools Qualification

The existing software aspects of airborne systems certification guidelines defined by the FAA through Advisory Circular 20-115B [1] and DO-178B [2] define software development tools as: *"Tools whose output is part of airborne software and thus can introduce errors"*. As elaborated in DO-178B and related documents [3,4], qualification is a supplementary process the applicant may elect to follow in a course of certification for the airborne system. Tool qualification is attempted only as an integral component of a specific certification program, i.e., part of a Type Certificate (TC), Supplemental Type Certificate (STC), or Technical Standard Order (TSO) approval. The tool data are referenced within the Plan for Software Aspects of Certification (PSAC) and Software Accomplishment Summary (SAS) documents for the original certification project. The applicant should present for review the Tool Operational Requirements (TOR) – a document describing tool functionality, environment, installation, operation manual, development process, expected responses (also in abnormal conditions). Two other documents must be submitted: Tool Qualification Plan and Tool Accomplishment Summary. To make an argument for qualification, the applicant must demonstrate that the tool complies with its TOR. This demonstration may involve a trial period during which a verification of the tool output is performed and tool-related problems are analyzed, recorded and corrected. The document states also that software development tools should be verified to check the correctness, consistency, and completeness of the TOR and to verify the tool against those requirements. More data is required for the qualification of a development tool, including Tool Configuration Management Index, Tool Accomplishment Summary, Tool Development Data, Tool Verification Data, Tool Quality Assurance Records, Tool Configuration Management Records, etc. These requirements are described in chapter 9 of the FAA Order 8110.49 [4].

Currently, there is no central repository that maintains a listing of previously qualified tools. Only the applicant, who qualified a tool within the scope of a specific project, has

or owns the necessary data. The conducted survey identified only a handful of development tools that have been qualified. In addition, the obtained information is anecdotal based on personal contacts and word of mouth rather than documented in a way that could be examined in detail. The only information the research team was able to obtain were statements to the effect: "Tool X was qualified while used on Level-Y certified system Z by applicant W". Occasionally, additional information was conveyed in a form of brief e-mails discussing the qualification approach or other details. More often the follow-up was unanswered. Efforts to get an access to specific documentation were unsuccessful. We may hypothesize that main reasons for that are intellectual property rights, company policies guarding competitive advantage, and the business red tape.

One needs to note that modern commercial development tools are typically complex suites combining multiple functionalities/capabilities. The survey identified a relatively short list of qualified development tools, or more specifically: selected functionality of the tool suite. At the time of this writing they included code generators (GALA: Generation Automatique de Logiciel Avionique, GPU: Graphical Processing Utility, VAPS CG: Virtual Application Prototyping System Code Generator, SCADE QCG: Safety Critical Avionic Development Environment Qualifiable Code Generator) and configuration-scheduling table generators (UTBT: Universal Table Builder Tool, CTGT: Configuration Table Generation Tool), most of them being in-house products.

Assessment Criteria and Experiments

In order to help in the tool qualification process, we first attempted to identify criteria (metrics) for tool evaluation, establish their measures, and conduct experiments validating the approach. Of the multiple criteria that have been used for tool evaluation in the past, we selected the following four [5], listed here with their measures:

- *Usability* measured as development effort (in hours)
- *Functionality* measured via the questionnaire (on a 0-5 points scale)
- *Efficiency* measured as code size (in LOC)
- *Traceability* measured by manual tracking (in number of defects).

This selection was based on the analysis of documentation from the FAA, ISO/IEC standards, industry survey, and studies done by the British Computer Society and Technical Center of Finland.

The experiments that followed were conducted on a generic avionics system test-bed equipped with a flight simulator, with real industry-strength software design tools [5]. Data were collected in four phases of the design process including:

(1) Project preparation
(2) Model creation and code generation
(3) Actual measurement
(4) Postmortem.

The recently published results [6] confirm that this approach can be effectively used to distinguish between tools within the group of established criteria.

Selected Additional Observations

- Despite their diversity, complex design tools with analysis and code generation capability dominate software tools market. Software for civil aviation systems is risk averse and provides a low quantity market not having enough commercial clout to drive the software tool market. On the other hand, use of tool on specific safety-critical project is a good public relation opportunity for tool vendor, which may result in increase of sales in less-regulated industries.
- Modern complex multifunctional tools require a steep learning curve. Considering tool complexity, the quality of support materials is often marginal. Unless developers become expertly proficient with the tool, reliance on it may lead to ignorance of tool functionality and complacency.
- The notation used by specific tool constraints the design options, thus restricting design flexibility. The tools may exhibit behavioral discrepancy due to the underlining runtime model.
- No mechanism exists to promulgate information about tool qualification. The qualification data constitute component of the certification package and are highly proprietary.
- It is imperative that the objectives for development tool qualification reflect the fact that the modern tools operate in an environment different than the target system. The typical operating environment for a tool is a general-purpose COTS workstation under conventional operating system. The critical issue for the tools is the integrity of the data as opposed to the tool operation in terms of timing, memory use, etc.
- Service history provides a means for claiming partial certification credits for target software. It does not help greatly to provide means for the development tools qualification, due to rapid progress of software technology. Typically, by the time

enough data is collected to create appropriate service history, the tool has been updated or modified in some way. Thus, in general, there is not enough time to get service history data for a development tool.

Conclusion

The experiments and observations indicate that the industry might benefit from methods to qualify a tool that are independent of a specific program and the applications using it. This would require updating the guidelines to consider that the tools operate in ground-based COTS environment different from the target application. This would also require considering a model-based development paradigm, re-defining the qualification process and allowing flexibility regarding qualification that is less dependent on the application program using the tool. A service history approach, considering incremental tool changes, may be also needed. A more streamlined method to qualify development tools and to keep them current as technology advances would be also useful. The streamlining must, however, not compromise safety.

References

[1] U.S. Dept. of Transportation, Federal Aviation Administration, *Advisory Circular AC 20-115B*, November 1993

[2] Radio Technical Commission for Aeronautics, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA DO-178B, RTCA SC-167, 1992

[3] Radio Technical Commission for Aeronautics, *Final Report for Clarification of DO-178B 'Software Considerations in Airborne Systems and Equipment Certification'*, RTCA DO-248B, RTCA SC-190, 2001

[4] U.S. Department of Transportation, Federal Aviation Administration, *Software Approval Guidelines, FAA Order 8110.49, (Chapter 9 replaces FAA Notice N8110.91 of 2001)*, FAA, 2003

[5] Kornecki A., Zalewski J., *Process Based Experiment for Design Tool Assessment in Real-Time Safety-Critical Software Development*, Proc. SEW-29, Annual 29th NASA/IEEE Software Engineering Workshop, Greenbelt, MD, April 6-7, 2005

[6] Kornecki A., J. Zalewski, *Experimental Evaluation of Software development Tools for Safety-Critical Real-Time Systems*, Innovations in Systems and Sofware Engineering – A NASA Journal, Vol. 1, No. 2, October 2005

A Method for Verification and Validation Certificate Management in Eclipse

Mark Sherriff
North Carolina State University
Raleigh, NC, USA 27695
+1-919-513-5082
mark.sherriff@ncsu.edu

Laurie Williams
North Carolina State University
Raleigh, NC, USA 27695
+1-919-513-4151
williams@csc.ncsu.edu

ABSTRACT
During the course of software development, developers will employ several different verification and validation (V&V) practices with their software, but these efforts might not be recorded or maintained in an effective manner. Our research objective is to build a method which allows developers to track and maintain a persistent record of the V&V practices used during development and testing. The persistent record of the V&V practices are recorded as certificates which are automatically stored and maintained with the code and creates traceability from the V&V practices to the code We have created a system that aids developers in the management of certificates in the Eclipse development environment. Also, we are researching a method to utilize a parametric model in conjunction with this V&V information to estimate the defect density of that program.

Categories and Subject Descriptors
D.2.8 **[Software Engineering]:** Metrics - *Performance measures*, *Process metrics*, *Product metrics.*

General Terms
Measurement, Design, Reliability

Keywords
Software Reliability Engineering, Reliability Estimation, Validation and Verification Management

1. INTRODUCTION
During software development, a development team will use several different methods to ensure that a system is of high-assurance [14]. However, the verification and validation (V&V) practices used to make a system reliable might not always be documented effectively or this documentation may not be maintained properly. This lack of proper documentation can hinder other developers from knowing what V&V practices have been performed on a given section of code. Further, if code is being reused from an earlier project or code base, developers might spend extra time re-verifying a section of code that has already been verified thoroughly.

One way to improve the documentation and management of V&V efforts is through the creation of certificates associated with the code base. *Certificates* are a record of a V&V practice employed by developers and can be used to support traceability between code and the evidence of the V&V technique used. These certificates can be automatically created, maintained, and verified by software tools, which allows developers to utilize them without excessive overhead.

Our research objective is to build a method which allows developers to track and maintain a certificate-based persistent record of the V&V practices used during development and testing. Further, we will build a parametric model which utilizes these certificates to provide an estimate the defect density of a program. To accomplish this objective, we are developing and automating a method called **D**efect **E**stimation with **V**&V **C**ertificates **o**n **P**rogramming (DevCOP). This method includes: (1) a mechanism for creating a persistent record of V&V practices as **certificates** stored with the code base; (2) **tool support** to make this method accessible for developers; and (3) a **parametric model** to provide an estimate of defect density.

We are currently developing a plugin for the Eclipse[1] integrated development environment (IDE) to support certificate management. The DevCOP Eclipse plugin allows developers to create, manage, and store certificates with the code base itself. We are also developing the DevCOP parametric model to provide an estimate of defect density using a nine-step systematic methodology for building software engineering parametric models based on work developed at the Center for Software Engineering at the University of Southern California [2, 11]. Research has shown that parametric models [5] using software metrics, such as the Software Testing and Reliability Early Warning (STREW) [8, 12] suite, can be an effective means to predict product quality. Due to the increasing cost of correcting defects during the software development lifecycle, developers can benefit from early information regarding the defect density of their product.

In this paper, we describe our current work in developing and validating the DevCOP parametric model and the DevCOP Eclipse plugin to support the creation and maintenance of DevCOP certificates.

2. BACKGROUND
In this section, we will discuss the relevant background work and methodologies used during our research, including metric-based defect density estimation; V&V techniques; and parametric modeling in software engineering.

2.1 Parametric Modeling
Parametric models relate dependent variables to one or more independent variables based on statistical relationships to provide an estimate of the dependent variable with regards to previous data [5]. The general purpose of creating a parametric model in software engineering is to help provide an estimated answer to a software development question early in the process so that development efforts can be directed accordingly. The software development question could relate to what the costs are in creating a piece of software, how reliable a system will be, or any number of other topics.

Parametric modeling has been recognized by industry and government as an effective means to provide an estimate for project cost and software reliability. The US Department of

[1] For more information, go to http://www.eclipse.org/.

Defense, along with the International Society of Parametric Analysts, acknowledges the benefit of using parametric analysis, and encourages their use when creating proposals for the government [5]. The Department of Defense claims that parametric modeling has reduced government costs and also improved proposal evaluation time [5].

Boehm developed the Constructive Cost Model (COCOMO) [3] to estimate project cost, resources, and schedule. Further, the Constructive Quality Model (COQUALMO) added defect introduction and defect removal parameters to the COCOMO to help predict potential defect density in a system. Nagappan [8] created a parametric model with his Software Testing Reliability Early Warning (STREW) metric suite to create an estimate of failure density based on a set of software testing metrics. In our research, we will also build a parametric model to estimate defect density based upon V&V certificates recorded with the code.

2.2 Verification and Validation Techniques

During the creation of software, a development team can employ various V&V practices to improve the quality of the software [1]. For example, different forms of software testing could be used to validate and verify various parts of a system under development. Sections of code can be written such that they can be automatically proven correct via an external theorem prover [14]. A section of a program that can be logically or mathematically proven correct could be considered more reliable than a section that has "just" been tested for correctness.

Other V&V practices and techniques require more manual intervention and facilitation. For instance, formal code inspections [4] are often used by development teams to evaluate, review, and confirm that a section of code has been written properly and works correctly. Pair programmers [15] benefit from having another person review the code as it is written. Some code might also be based on technical documentation or algorithms that have been previously published, such as white papers, algorithms, or departmental technical reports. These manual practices, while they might not be as reliable as more automatic practices due to the higher likelihood of human error, still provide valuable input on the reliability of a system.

The extent of V&V practices used in a development effort can provide information about the estimated defect density of the software prior to product release. The Programatica team at the Oregon Graduate Institute at the Oregon Health and Science University (OGI/OHSU) is working on a method for high-assurance software development [14]. Programmers can create different types of *certificates* on sections of code based on the V&V technique used by the development on that section of the code. Certificates are used to track and maintain the relationship between code and the evidence of the V&V technique used. Currently, the three types of V&V techniques that Programatica can create certificates for include expert opinion, unit testing, and formal proof. These certificates are used as evidence that V&V techniques were used to make a high-assurance system [14]. We propose an extension of OGI/OHSU's certificates for defect density estimation whereby the estimate is based upon the effectiveness of the V&V practice for identifying defects (or lack thereof) used in code modules.

2.3 Metrics to Predict Defect Density

Operational profiles have been shown to be effective tools to guide testing and help ensure that a system is reliable [6]. An operational profile is "the set of operations [available in a system] and their probabilities of occurrence" as used by a customer in the normal use of the system [7]. However, operational profiles are perceived to add overhead to the software development process as the development team must define and maintain the set of operations and their probabilities of occurrence. Rivers and Vouk recognized that operational profile testing is not always performed when modern constraints on market and cost-driven constraints are introduced [9]. They performed research on evaluating non-operational testing and found that there is a positive correlation between field quality and testing efficiency. Testing efficiency describes the potential for a given test case to find faults at a given point during testing. Our research uses non-operational methods to avoid excessive overhead, while still providing valuable information.

Nagappan [8] performed research on estimating failure density without operational profiles by calibrating a parametric model which uses in-process, static unit test metrics. This estimation provides early feedback to developers so that they can increase the testing effort, if necessary, to provide added confidence in the software. The STREW metric suite consists of static measures of the automated unit test suite and of some structural aspects of the implementation code. Case studies [8] indicate that the STREW-J metrics can provide a means for estimating software reliability when testing reveals no failures. Another version of the STREW metric suite was developed specifically for the Haskell programming language, STREW-H [12, 13]. STREW-H was similarly built and verified using case studies from open-source and industry. These findings also showed that in-process metrics can be used as an early indicator of software defect density for Haskell programs. In our research, we use a similar approach to predict defect density, taking software metrics and using a parametric model to provide early defect feedback to developers

3. THE DevCOP ECLIPSE PLUGIN

We have created a DevCOP Eclipse plugin to handle the creation and management of V&V certificates during the development process[2] [10, 13]. The main purpose of the plugin is to automate the DevCOP method with little additional overhead for developers. The plugin allows developers to create and store certificates during the development process within the IDE so that this information can be utilized throughout the code's lifetime for defect density estimation purposes, for maintenance purposes, for analysis of the effectiveness of certain V&V practices, or for future reference in reused code. Figure 1 shows a screenshot of the Eclipse plugin for recording V&V certificates.

The current version of the plugin, Version 1.1.1, focuses on recording certificates that normally do not produce artifacts that are stored with the code. In other words, the plugin will aid developers by recording information about manually-performed V&V, not automatic or programmatic V&V, such as unit testing. Programmers can select one or more functions for certification through the Eclipse Package Explorer. They select the type of certificate (i.e. Code Inspection, Pair Programming, Bug Fix) and the relative importance of the certificate as the weight coefficient associated with it. The certificate information is then stored in an XML document that is saved in the project's workspace. The Eclipse plugin reads and writes to these XML documents as certificates are created and edited.

The example certificate shows the information that is recorded about the V&V technique and the function on which the technique was used. Basic identifying information, such as the name of the

[2] The plugin is available at http://agile.csc.ncsu.edu/mssherri/devcop/.

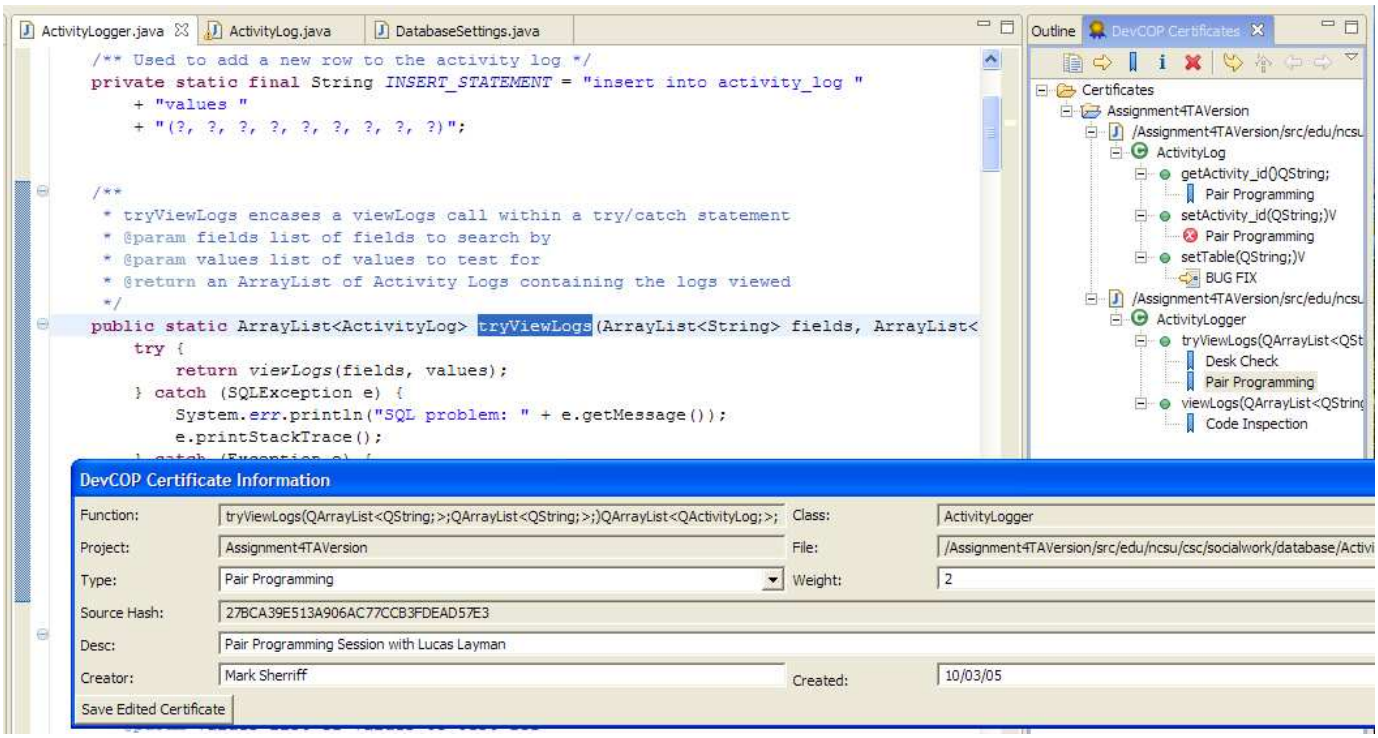


Figure 1. Screenshot of the DevCOP Eclipse plugin for recording V&V certificates.

function and its location along with creation information, are all included. At the time of creation, the certificate stores a hash of the source code with the certificate. The plugin determines whether a certificate is valid or not by comparing a stored hash of the source code at the time of certification with that of the current source code. If a change is made to a function's code, the source hash stored with the certificate no longer matches that of the function's source code, and the certificate is invalidated.

Our objective is to make the certificate creation process as easy and transparent as possible, and will continue to improve it in later iterations as we receive more developer feedback. The primary method in which we accomplish this method of certificate creation is through what we call Active Certificates. An *Active Certificate* is a means by which Eclipse will automatically identify changed code during a programming session to be certified by the developer.

For example, if two programmers were about to start pair programming on a piece of code, they would click the Active Certificate button before they began. Eclipse would then actively record non-trivial changes to the system (i.e. changes to the abstract syntax tree of the code, not commenting or formatting changes) and will present the affected functions to the developers for certification at the end of the pair programming session. The concept of Active Certificates can extend to several different types of V&V activity, such as code inspections or bug fixes. Active Certificates allow developers to write or modify code normally, without increasing their work overhead.

Current improvements are being made to the plugin based on feedback from development teams. Suggestions such as integrating the DevCOP plugin with the refactoring mechanism in Eclipse or with the source control system itself are being considered. The automatic creation of other types of certificates is another feature under development. Certificates will be dynamically created from automated testing suites or other similar methods of V&V. We are also adding reporting functionality to the plugin to provide developers tools for evaluating their overall V&V effort, including identifying sections that might not have been covered by any V&V techniques as of yet.

4. THE DevCOP PARAMETRIC MODEL

A V&V certificate in DevCOP contains information on the V&V technique that was used to establish the certificate and is associated with a specific function in a piece of code. Different V&V techniques will provide a different level of assurance as to how reliable a section of code is. For example, a desk check of code would be, in general, less effective than a formal proof of the same code.

We are developing a parametric model which uses non-operational metrics to estimate defect density based upon records of which V&V practices were performed on sections of code during development [10]. We also wish to integrate our estimation directly into the development cycle so that developers may take corrective measures earlier in the development lifecycle.

We envision the defect density parametric model to take the form of Equation 1. For each certificate type, we would sum the product of a size measure (perhaps lines of code or number of functions/methods) and a coefficient produced via regression analysis of historical data. The calibration step of the regression analysis would yield the constant factor (a) and a coefficient weighting (c_j) for each certificate type, indicating the importance of a given V&V technique to an organization's development process.

$$DefectDensity = a + \sum_{j=1}^{certificate\_type} (c_j * Size_j) \quad (1)$$

To build and verify our parametric model of our DevCOP method, we are utilizing the nine-step modeling methodology [11]:

1. Determine model needs;
2. Analyze existing literature;
3. Perform behavioral analysis;
4. Define relative significance;
5. Gather expert opinion;
6. Formulate a priori model;
7. Gather and analyze project data;
8. Calibrate a posteriori model; and
9. Gather more data; refine model.

The goal of the model is to provide an estimate of defect density based on V&V certificates and the coverage of each certificate type. We anticipate that a model would need to be developed for each programming language we would study. Our current work involves the Java (object-oriented) and Haskell (functional) languages. We are currently investigating how this technique can be applied to these two different languages.

5. LIMITATIONS

In the creation of certificates, we are not assigning more importance to certain functions or sections of code over others, as is done with operational profile means of estimation. Nor are we using the severity of defects detected to affect the importance of some certificates over another. While this level of granularity could be beneficial, one of our initial goals is to make this method easy to use during development, and at this time, we think that adding this level of information could be a hindrance. Another limitation is the granularity of certificates. Based on the Programatica Team's work [14] it was decided that methods would be the proper level of granularity for certificates. The determination of certificate weights used in the parametric model is still being researched through empirical studies with industry projects.

6. CONCLUSIONS AND FUTURE WORK

We have created and are currently validating a method for managing V&V certificate information. We are also developing a method for a development team to estimate software defect density in-process using this V&V information. Due to the high costs of fixing software defects once a product has reached the field, information that can be provided to developers in-process and can give an indication of software defect density is invaluable. If corrective actions can be taken earlier in the software development life cycle to isolate and repair software defects, overall maintenance costs can decrease.

The DevCOP plugin allows developers to easily record their V&V activities within the development environment without increasing their overhead greatly due to the inclusion of Active Certificates. The plugin also provides developers with a mechanism to manage the effort that is put into V&V in a place where all developers can see what measures have been taken to ensure a piece of code is reliable and to treat it accordingly. DevCOP certificate information can be used to provide a V&V history for particular code segments. Further, after a set of certificates has been created, an overall estimate of defect density can be created based on the V&V weightings using a parametric model. We will continue our work to improve the plugin based on developer suggestions and to gather data to validate the DevCOP parametric model.

7. ACKNOWLEDGEMNETS

We wish to give our sincerest thanks to the Programatica team for their input on the various parts of this work. This work was funded by the National Science Foundation.

8. REFERENCES

[1] Balci, O., "Verification, Validation, and Accreditation of Simulation Models," *Winter Simulation Conference*, 1997, pp. 125-141.

[2] Boehm, B. W., "Building Parametric Models," *International Advanced School of Empirical Software Engineering*, Rome, Italy, September 29, 2003.

[3] Boehm, B. W., Horowitz, E., Madachy, R., Reifer, D., Clark, B., Steece, B., Brown, A. W., Chulani, S., and Abts, C., *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall, 2000.

[4] Fagan, M., "Design & Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1979.

[5] International Society of Parametric Analysts, "Parametric Estimating Handbook." Available Online. Online Handbook. http://www.ispa-cost.org/PEIWeb/Third_edition/newbook.htm.

[6] Musa, J., "Theory of Software Reliability and its Applications," *IEEE Transactions on Software Engineering*, pp. 312-327, 1975.

[7] Musa, J., *Software Reliability Engineering*: McGraw-Hill, 1998.

[8] Nagappan, N., "A Software Testing and Reliability Early Warning (STREW) Metric Suite," PhD Dissertation, North Carolina State University, 2005.

[9] Rivers, A. T., Vouk, M.A., "Resource-Constrained Non-Operational Testing of Software," *International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998, pp. 154-163.

[10] Sherriff, M., "Using Verification and Validation Certificates to Estimate Software Defect Density," *Doctoral Symposium, Foundations of Software Engineering*, Lisbon, Portugal, September 6, 2005, 2005.

[11] Sherriff, M., Boehm, B. W., Williams, L., and Nagappan, N., "An Empirical Process for Building and Validating Software Engineering Parametric Models," North Carolina State Univeristy CSC-TR-2005-45, October 19 2005.

[12] Sherriff, M., Nagappan, N., Williams, L., and Vouk, M. A., "Early Estimation of Defect Density Using an In-Process Haskell Metrics Model," *First International Workshop on Advances in Model-Based Software Testing*, St. Louis, MO, May 15-21, 2005.

[13] Sherriff, M., Williams, L., "Tool Support For Estimating Software Reliability in Haskell Programs," *Student Paper, IEEE International Symposium on Software Reliability Engineering*, St. Malo, France, 2004, pp. 61-62.

[14] The Programatica Team, "Programatica Tools for Certifiable, Auditable Development of High-Assurance Systems in Haskell," *High Confidence Software and Systems*, Baltimore, MD, 2003.

[15] Williams, L. and Kessler, R., *Pair Programming Illuminated*. Boston: Addison-Wesley, 2002.

Certificate Management: A Practitioner's Perspective

Mike Whalen, Rockwell-Collins Inc.
mwwhalen@rockwellcollins.com

Standards for critical avionics software development, such as DO178B, place a strong emphasis on process issues: ensuring traceability between different development artifacts and proper configuration management of these artifacts. Certification Management (CM) systems formalize many of the relationships between different artifacts and hold the promise of both streamlining the management of the artifacts and ensuring that relationships between the artifacts are formally justified. However, to be useful in an industrial context, the definition and scope of CM systems must be better understood, and several open issues must be addressed. This paper describes issues and potential uses of CM systems in industrial practice.

1. Introduction

Current avionics software standards such as DO178B [6] focus on software development processes to try to ensure a high level of confidence in the correctness of the developed software. These process requirements include ensuring traceability between requirements, design artifacts, source, and object code, and also in maintaining proper configuration management between artifacts. However, little emphasis is placed on formal verification of functional behavior of systems.

Recent advances in modeling languages have made it feasible to formally specify and analyze the behavior of large system components. Synchronous data flow languages, such as Lustre [1], SCR [2], and RSML^{-e} [3] seem to be particularly well suited to this task, and commercial tools such as SCADE [4] and Simulink [5] are growing in popularity among designers of safety critical systems, largely due to their ability to automatically generate code from models. At the same time, advances in formal analysis tools have made it practical to formally verify important properties of models to ensure that design defects are identified and corrected early in the lifecycle (see, for example, [8], [9], [10]). At Rockwell-Collins we are integrating formal analysis into the design and development cycle for next-generation commercial avionics systems and expect formal analysis to be an integral part of the V & V process for future systems.

Software certification management (CM) systems are designed to support independent verification of some aspect of software development. They introduce the notion of a *software certificate*, which contains all the information necessary for an independent assessment of the demonstrated properties. These certificates could be used to formalize many of the analyses that are required in guidelines such as DO178B, and also for formal functional verification of software artifacts, leading to safer systems.

Unfortunately, the current definition of CM systems is diffuse, and it is difficult to determine the boundaries between CM systems, configuration management systems such

as CVS and Rational ClearCase, and requirements traceability systems such as DOORS. When managing informal artifacts, such as, for example, fault trees, textual requirements, or design rationales, the benefit of using a CM approach over traditional traceability tools is unknown. In order to use CM systems in an industrial setting, a more specific definition of the role and benefit of CM systems is required. This paper presents a few thoughts on how and where CM systems might be useful in a critical software development effort, and some future directions for research.

2. CM Opportunities for Showing Safety and Requirements Traceability in Critical Avionics Software

Critical software standards such as DO178B [6] require multiple levels and types of traceability between software artifacts. It distinguishes four abstraction layers of software artifacts: *high-level requirements, low-level requirements, source code,* and *object code,* and requires that the artifacts in each layer map to artifacts in the preceding and proceeding layer. The standard approach to satisfying DO178B uses a mixture of semi-formal analysis (often consisting of human inspections and checklists) and extensive testing to try to show that software is correctly implemented and corresponds to its requirements.

DO178B calls out several different kinds of analysis that should be performed on source code. Some of these analyses are designed to show conformance to higher-level requirements, while others are "well-formedness" checks to ensure that implementation does not allow safety or security violations. Many of the well-formedness criteria could be easily formulated as Proof-Carrying Code (PCC)-style safety policies to be proven of source or object code. These proofs could then be used as certificates for the system in question. Some well-formedness properties that are called out in DO178B are:

- unit-of-measurement/dimensional consistency between modules / subsystems
- arithmetic overflow/underflow
- variable initialization-before-use
- behavior of partial arithmetic operators (e.g. divide)
- termination
- deadlock/livelock/race conditions
- array/pointer safety

There are several tools that can automatically check such properties, such as PolySpace [13], but these currently do not generate evidence suitable for CM systems.

With the recent adoption of model-based development languages such as SCADE [1] and Simulink [5], it has also become easier to formally analyze functional behavior of software. These languages have relatively straightforward formal semantics that are straightforward to translate into model checking languages such as SMV [7]. Rockwell-

Collins has had significant success translating informal textual requirements into properties that can be proven against large software models [10].

In order to perform the proofs, it was necessary to split the software model into several *analysis models* and use techniques such as *temporal induction* [11] to perform assume-guarantee proofs. These creation of the analysis models and the *proof graphs* [11] were performed and justified by hand. In order to perform automated formal analysis of large systems, these kinds of steps will be necessary, and the hand-justifications are a weak link in the guarantees provided by the model checking tools. Certificates that ensure that the different analyses are correctly justified and related would be a significant benefit.

Another issue is that analyses performed by most model checking tools do not yield any kind of certificate, so cannot be justified. This leads to a situation where a significant amount of trust must be invested in the model checker. Another avenue for improvement would be the creation of certificate-generating automated analysis tools.

3. What is a Certificate Management System?

In order to use CM systems in avionics projects, we must better define what they are. According to the SoftCeMent web site, these CM systems look suspiciously like end-to-end CASE tools that require significant commitment of resources, including functions such as configuration management tools, databases, traceability tools, make tools, workflow tools, and audit and reporting tools. Businesses already have mature, established tools for most of these tasks, and it is unlikely that they will switch over to a single system for managing all of this functionality.

Also, there seems to be some fuzziness on what certificates are and how much assurance they can provide. Given a formal proof of some safety property on source or object code and the code itself, one can derive a very high level of confidence by mechanically checking the proof. On the other hand, given an informal safety property and an informally generated fault tree, what kind of guarantees can a certificate provide? In this case, it is difficult to see what benefit CM systems would provide over a simple code-signing approach provided by component tools such as Microsoft's .NET assemblies or Java JAR files.

We would suggest that most of the interesting and beneficial features of CM can be hosted as relatively self-contained "plug-ins" to existing tools. A tool like DOORS already has sophisticated traceability, linking, and reporting facilities. Plug-ins could be created to both help generate and check whether formal relationships hold between two artifacts within the database. Just this aspect of certificate management presents an enormous challenge, as there is a wide range of logics that can be used to check these properties and potential formats for certificates, not to mention challenges in automating the generation of certificates. It seems unnecessary and unwise to try to tackle software management issues that are capably handled by existing tools.

4. Conclusion

Formal tools and techniques are increasingly used in practice to help create and verify the behavior of critical software systems. Certification management systems have the

potential to significantly streamline and formalize many of analyses that are required in avionics standards such as DO178B [6]. However, tools to support certification management are still in their infancy. To see significant industrial adoption, they must be fairly easy to use and integrate with existing configuration management and traceability tools. In order to be widely adopted, these tools must also integrate well into a certification story that is acceptable to authorities such as the FAA, since they ultimately decide whether a system is airworthy.

References

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R.de Simone, The Synchronous Languages 12 Years Later, Proceedings of the IEEE, Volume 91, Issue 1, January 2003.

[2] C. Heitmeyer, R. Jeffords., and B. Labaw, Automated Consistency Checking of Requirements Specification, ACM Transactions on Software Engineering and Methodology (TOSEM), 5(3):231-261, July 1996.

[3] J. Thompson, M. Heimdahl, and S. Miller.: Specification Based Prototyping for Embedded Systems, Proceedings of the Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering, LNCS 1687, September 1999.

[4] Esterel Technologies, http://www.esterel-technologies.com.

[5] James Dabney and Thomas Harmon, Mastering Simulink, Pearson Prentice Hall: Upper Saddle River, NJ, 2004.

[6] RTCA. Software Considerations In Airborne Systems and Equipment Certification (DO178B), RTCA, 1992

[7] IRST, http://nusmv.irst.itc.it/, The NuSMV Model Checker, Trento Italy

[8] Marco Bozzano, Antonella Cavallo, Massimo Cifaldi, Laura Valacca, and Adolfo Villafiorita, Improving Safety Assessment of Complex Systems : An Industrial Case Study. Proceedings of Formal Methods 2003 (LNCS 2805), Springer-Verlag, pages 208-222, 2003.

[9] R. Butler, S. Miller, J. Potts, and V. Carreno, A Formal Methods Approach to the Analysis of Mode Confusion, Proceedings of the 17th AIAA/IEEE Digital Avionics Systems Conference, Bellevue, WA, October 1998.

[10] S. P. Miller, M. P.E. Heimdahl, and A.C. Tribble, Proving the Shalls, Proceedings of FM 2003: the 12th International FME Symposium, Pisa, Italy, Sept. 8-14, 2003.

[11] K. L. McMillan, Circular Compositional Reasoning About Liveness, Cadence Berkeley Labs Technical Report 1999-02, Berkeley, CA, 1999.

[12] Telelogic, Inc. DOORS product description web page. http://www.telelogic.com/products/doorsers/index.cfm

[13] PolySpace Technologies, Inc. PolySpace product description web page. http://www.polyspace.com

CERTIFYING SOFTWARE FIT FOR PURPOSE

Graeme Parkin and Peter Harris,
National Physical Laboratory

1 Introduction

The use of software in measurement systems has dramatically increased over the last few years, making devices easier to use, more reliable and more accurate. However the hidden complexity within the software is a potential source of undetected errors. Since it is hard to quantify the reliability or quality of such software, two questions arise:

- As a user of such a system, how can I be assured that the software is of sufficient quality to justify its use?
- As a supplier of such software, what validation techniques should I use, and how can I assure my users of the quality of the resulting software?

A means to certify that software is fit for purpose is required by both users and suppliers of measurement systems. It is not possible to test software exhaustively. There are many examples reported in the public domain of errors in software that have been very costly, either in money or life. For example, the Ariane 5 launcher ended in failure, the launcher veered off its flight path, broke up and exploded costing $370 million, due to a wrong conversion of a 64 bit value[1]. So even when best practice has been applied software can still have bugs. There are many possible techniques that can be applied in the development of software to reduce the number of errors. However the application of these techniques costs both time and money with diminishing returns.

An approach is described which determines which techniques should be used to produce software fit for purpose. This is illustrated by an example. It is also explained why instrument manufacturers are interested in this work for certifying products for safety-critical applications.

2 A solution

A risk analysis approach is taken to determine the techniques to be applied in the development of software which is fit-for-purpose. The risk analysis is based on three parameters, criticality of usage, complexity of processing and complexity of control, to which values are assigned. Each parameter can take one of four values. Criticality of usage values are one of critical, business critical, potentially safety-critical and safety-critical. Complexity of processing values are one of very simple, simple, moderate and complex. Complexity of control values are one of very simple, simple, moderate and complex. A further consideration is any legal obligations that may have to be met. Having assigned values to the risk parameters a Measurement Software Level (MSL) is determined based on Table 1. Having calculated a MSL, Table 2 is used to determine the techniques to develop the software so that it is fit-for-purpose. The Guide assumes a quality system is in place e.g ISO 9000 series of standards[2].

3 Application

In a recent application of the approach it was required to produce reference software for the calculation of surface texture parameters based on a profile[3] and be able to read profile data in SMD format[4]. The software was also required to work across platforms and give the same results on each. An example of a parameter is shown in Figure 1. Figure 2 shows briefly the derivation of the MSL, the techniques to be used to meet that MSL and the tools used. Other tools used were an IDE (BlueJ 2.0.3), component testing (JUnit 3.8.1) and a Java-based build tool (Ant 1.6.2).

[1] N° 33-1996: Ariane 501 - Presentation of Inquiry Board report.
[2] ISO 9001 2000: Quality management systems -- Requirements, ISO IEC 90003 2004: Software engineering - Guidelines for the application of ISO 9001:2000 to computer software.
[3] ISO 4287 Geometrical Product Specifications (GPS) -- Surface texture: Profile method -- Terms, definitions and surface texture parameters. 1997.
[4] ISO 5436-2 Geometrical Product Specifications (GPS) -- Surface texture: Profile method; Measurement standards -- Part 2: Software measurement standards. 2001.

4 The guide

The process outlined in the previous sections is much more fully described in Best Practice Guide No1, Validation of Software in Measurement Systems[5]. The guide has been designed to be used as the basis of certification services mainly with auditable checklists.

Criticality of usage	Complexity of Processing	Impact of complexity of control			
		Very simple	Simple	Moderate	Complex
Critical	Very simple	0	0	1	2
	Simple	0	1	1	2
	Moderate	1	1	2	2
	Complex	2	2	2	2
Business Critical	Very simple	0	1	1	2
	Simple	1	1	2	2
	Moderate	1	2	2	2
	Complex	2	2	2	3
Potentially life-critical	Very simple	1	1	2	2
	Simple	1	2	2	3
	Moderate	2	2	3	3
	Complex	2	3	3	3
Life-critical	Very simple	2	2	2	3
	Simple	2	2	2	3
	Moderate	2	2	3	4
	Complex	3	3	4	4

Table 1 Measurement Software Level as function of risk factors (see Guide for further details)

Furthermore the guide, when used for safety-critical software, assists compliance with the international standard for functional safety IEC 61508. The guide is to be used as input to determining a means to certify products to IEC 61508 by the 61508 Association[6] which was set up by instrument manufacturers in the UK. Currently the guide is being used to evaluate the software in alarm annunciators for Evaluation International[7].

5 Development of the guide

The guide was designed to provide advice which would satisfy a range of standards including: ISO/IEC 17025[8], Legal metrology[9], IEC 601-1-4[10], IEC 61508[11] and DO-178B[12]. The techniques

[5] Software Support for Metrology, Best Practice Guide No. 1, Validation of Software in Measurement Systems Brian Wichmann, Graeme Parkin and Robin Barker March 2004, Version 2.1, http://www.npl.co.uk/ssfm/download/documents/ssfmbpg1.pdf (freely available).
[6] http://www.61508.org.uk/
[7] http://www.evaluation-international.com/
[8] ISO/IEC 17025: 2005. General requirements for the competence of testing and calibration laboratories.
[9] WELMEC 2.3 Guide for examining software (Non-automatic weighing instruments), January 1995. WELMEC 7.1 Software requirements on the basis of the measuring instruments directive, January 2000. Both available at http://www.welmec.org/pubs.asp.
[10] IEC 601-1-4 Medical electrical equipment – Part 1: General requirements for safety 4: Collateral standard: Programmable electrical medical systems.
[11] IEC 61508: Parts 1-7, Functional safety of electrical/electronic/programmable electronic (E/E/PE) safety-related systems.

mentioned in the guide have been selected based on industry acceptance, tool support and ease of being audited. The guide has been reviewed, and their comments taken into account by persons in the following application areas of nuclear, medical, safety-critical and certification.

Ref.	*Recommended Technique*	*Measurement Software Level*			
		1	2	3	4
12.2	Review of informal specification	Yes	Yes		
12.3	Software inspection of specification		Yes	Yes	
12.4	Mathematical specification	Yes	Yes	Yes	Yes[13]
12.5	Formal specification				Yes[13]
12.6	Static analysis		Yes	Yes	Yes[13]
12.6	Boundary value analysis		Yes	Yes	
12.7	Defensive programming	Yes	Yes		
12.8	Code review	Yes	Yes		
12.9	Numerical stability		Yes	Yes	Yes[13]
12.10	Microprocessor qualification				Yes[13]
12.11	Verification testing			Yes	Yes[13]
12.12	Statistical testing		Yes	Yes	
12.13	Structural testing	Yes			
12.13	Statement testing		Yes	Yes	
12.13	Branch testing			Yes	Yes[13]
12.13	Boundary value testing		Yes	Yes	Yes[13]
12.13	Modified Condition/Decision testing				Yes[13]
12.15	Accredited testing		Yes		
12.16	System-level testing	Yes	Yes		
12.17	Stress testing		Yes	Yes	
12.18	Numerical reference results	Yes	Yes	Yes[13]	Yes[13]
12.19	Back-to-back testing		Yes	Yes	
12.20	Source code with executable				Yes[13]

Table 2 Recommended Techniques (see the guide for further details)

$$RSm = \frac{1}{m}\sum_{i=1}^{m} Xs_i$$

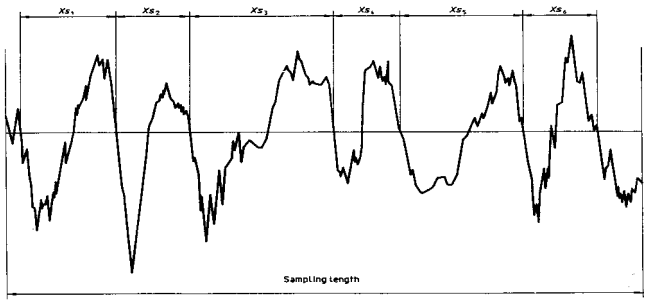


Figure 1 Spacing parameter RSm for a roughness profile

[12] DO-178B Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B). December 1992.
[13] These are still suggestions for MSL4 or, in the case of MSL3 are to be used if no alternative.

Risk analysis
- No legal requirements
- Business critical
- Simple complexity of control
- Moderate complexity of processing

(plus other issues like ease of testing etc.)

→ Measurement Software Level = 2

Ref.	*Recommended Technique*	*MSL 2*	*Used*	*How this is met*
12.2	Review of informal specification	Yes	Yes	-
12.3	Software inspection of specification	Yes	No	Based on international standard
12.4	Mathematical specification	Yes	Yes	MATLAB 7.0
12.5	Formal specification			Not applicable
12.6	Static analysis	Yes	Yes	Java compiler 1.4.2_4, Checkstyle 3.3
12.6	Boundary value analysis	Yes	Yes	-
12.7	Defensive programming	Yes	Yes	-
12.8	Code review	Yes	Yes	Checkstyle 3.3
12.9	Numerical stability	Yes	Yes	-
12.10	Microprocessor qualification			Not applicable
12.11	Verification testing			Not applicable
12.12	Statistical testing	Yes	No	-
12.13	Structural testing			Not applicable
12.13	Statement testing	Yes	Yes	Clover 1.3_02
12.13	Branch testing		Yes	Clover 1.3_02
12.13	Boundary value testing	Yes	Yes	-
12.13	Modified Condition/Decision testing			Not applicable
12.15	Accredited testing	Yes	No	Not applicable
12.16	System-level testing	Yes	Yes	-
12.17	Stress testing	Yes	Yes	Tested for large data sets
12.18	Numerical reference results	Yes	No	-
12.19	Back-to-back testing	Yes	Yes	Against MATLAB specifications
12.20	Source code with executable			Not applicable

Figure 2 Shows derivation of MSL and techniques used for the surface texture reference software

6 Summary

A means to certify software so that is fit for purpose has been briefly described. A service to certify software using the guide is being set up. Further work includes getting the guide more widely accepted, possibly through standardisation and developing guides on the use, application and evaluation of software development tools e,g, code coverage tools.

On the Scalability of Proof Carrying Code for Software Certification*

Andrew Ireland

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
a.ireland@hw.ac.uk

Abstract. Proof Carrying Code provides an approach to software certification, where trust management is decentralized. The approach has been successfully applied to relatively simple properties. Here we consider the scalability of the approach when more comprehensive properties are considered, *e.g.* functional properties. We argue that tactic-based theorem proving, and in particular proof plans, have a role to play in addressing the issue of scalability.

1 Introduction

Within the *Proof Carrying Code* (PCC) paradigm [7], software certificates correspond to formal proofs, *i.e.* a proof that a program satisfies a given safety policy. The responsibility for proof construction lies with the code producer, while a relatively light-weight proof checking process is all that is required on the consumer side. Moreover, the consumer does not need to trust the producer or any third-party intermediaries. As a consequence, PCC decentralizes trust management, *i.e.* the *trusted computing base* is minimal and local to the consumer.

Initially, proofs were relatively large, given the size of code involved. Significant progress, however, has been made in reducing the size of proofs, *i.e.* software certificates. In particular, an approach known as *Oracle-based Proof Carrying Code* (OPCC) uses *oracle strings* [8] as a means of representing the minimal information required for proof checking, *i.e.* the checker is only provided with information when a choice is required. Proof *tactics* have also been used to reduce the size of proofs, *i.e.* large proof steps defined in terms of tactics. This is known as *Tactic-based Proof Carrying Code* (TPCC) [1]. Of course within TPCC, the tactic definitions are required for proof checking, thus increasing the machinery on the consumer side.

PCC has been mainly concerned with safety properties, such as type safety and memory management safety. The relative light-weight nature of these properties has meant that proof construction corresponds to type inference. The need for more comprehensive properties is widely recognized. For instance, the MOBIUS project[1] has identified the need for comprehensive policies, such as functional properties, as one of the *"challenges that lie far beyond the current state-of-the-art"*. Meeting this challenge will increase the burden of proof associated with PCC, both in terms of proof construction and communication. Below we explore these issues in more detail.

2 Proof Construction

Extending PCC to include functional properties introduces all the complexities that are associated with software verification, *e.g.* the need for code to be annotated

* The work discussed was supported in part by EPSRC grant GR/S01771.

[1] MOBIUS: `http://mobius.inria.fr/twiki/bin/view/Mobius`.

with auxiliary assertions, such as loop invariants. The current focus on type-based methods will need to be combined with logic-based methods. In particular, theorem proving and program analyzers that assist with the generation of code annotations will be required.

We believe that the technique known as *proof planning* [3] also has a role to play here. Proof planning is a computer-based technique for automating the search for proofs. At the core of the technique are high-level proof outlines, known as *proof plans*. A proof plan embodies a *generic* tactic and is typically hierarchical in structure. Proof planning is the process by which a customized tactic is constructed for a given conjecture. The generic nature of a proof plan makes for a robust style of reasoning, *i.e.* proof planning can deal with changes to a conjecture, as long as the changes fall within the scope of the given proof plan. The use of proof planning to support proof construction would therefore represent a natural extension to TPCC. In terms of program analysis, proof planning has also demonstrated its value through the NuSPADE project[2], where proof planning was investigated within the context of verifying software written in SPARK [2]. In particular, proof-failure analysis, a key feature of proof planning, was used in conjunction with program analysis to guide the generation of loop invariants [4–6].

3 Proof Communication

As noted above, OPCC and TPCC have achieved significant reductions in the size of proofs. It is unclear, however, whether or not these approaches will scale to meet the challenges associated with more comprehensive properties. Here we propose an alternative approach. Instead of communicating a proof, or *how* to construct a proof (via a tactic or proof oracle), we propose communicating *what* knowledge is required in order for the consumer to re-construct a producer's proof. What we will refer to as *Proof Plan Carrying Code* (PPCC), can be viewed as an extension of TPCC. To achieve PPCC, we envisage the notion of a *Proof Planning Oracle* (PPO) , *i.e.* information on which proof plans and theories were used in planning a particular conjecture or class of conjectures. We see PPOs as an optional input/output to the existing proof planning framework. That is, the producer will use a proof planner to generate a PPO which is then used to constrain proof planning on the consumer side.

While PPOs will significantly reduce the size of software certificates, it will also significantly increase the burden on the code consumer. Firstly, the code consumer will require access to the proof plan and theory repositories referenced by the PPO – introducing the problem of managing distributed repositories. Secondly, the code consumer will be required to run a proof planner as well as a proof checker – increasing the consumer's computational overhead. Note however that the PPO will significantly reduce the search involved in re-constructing proofs on the consumer side. As is the case with TPCC, this additional overhead will exclude on-device proof checking. For many applications this would be a show-stopper, *e.g.* smart card applications with minimal resources. However, we believe that such applications will also rule-out on-device proof checking with respect to the more comprehensive properties that are currently being considered. So where on-device checking is not essential, but where comprehensive properties are mandatory, then PPCC may provide a practical approach.

[2] NuSPADE: `http://www.macs.hw.ac.uk/nuspade`.

4 Conclusion

PCC has been applied successfully to relatively simple properties. Targeting more comprehensive properties raises questions about the scalability of current approaches. We have argued that proof plans have a role to play in addressing the scalability of proof construction. In terms of representing software certificates, we have proposed the use of proof planning oracles as a technique for reducing the size of formal proofs.

References

1. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In *Proc. Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2005.
2. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.
3. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
4. B.J. Ellis and A. Ireland. Automation for exception freedom proofs. In *Proceedings of the* 18^{th} *IEEE International Conference on Automated Software Engineering*, pages 343–346. IEEE Computer Society, 2003. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0010.
5. B.J. Ellis and A. Ireland. An integration of program analysis and automated theorem proving. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of 4th International Conference on Integrated Formal Methods (IFM-04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 67–86. Springer Verlag, 2004. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0014.
6. A. Ireland, B.J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to program reasoning. Technical Report HW-MACS-TR-0027, School of Mathematical and Computer Sciences, Heriot-Watt University, 2004.
7. G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.
8. G.C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.

Software Certification for Temporal Properties with Affordable Tool Qualification

Songtao Xia and Ben Di Vito

Mail Stop 130
NASA Langley Research Center
Hampton, VA 23281
{s.xia, b.l.divito}@larc.nasa.gov

Abstract. It has been recognized that a framework based on proof-carrying code (also called semantic-based software certification in its community) could be used as a candidate software certification process for the avionics industry. To meet this goal, tools in the "trust base" of a proof-carrying code system must be qualified by regulatory authorities. A family of semantic-based software certification approaches is described, each different in expressive power, level of automation and trust base. Of particular interest is the so-called abstraction-carrying code, which can certify temporal properties. When a pure abstraction-carrying code method is used in the context of industrial software certification, the fact that the trust base includes a model checker would incur a high qualification cost. This position paper proposes a hybrid of abstraction-based and proof-based certification methods so that the model checker used by a client can be significantly simplified, thereby leading to lower cost in tool qualification.

1 Introduction

Safety critical programs, such as those controlling an airplane, a nuclear power plant, or a medical system, are subject to the highest level of verification and validation (V & V) effort, which is often outlined by administrative authorities. For example, to deploy an autopilot program onboard an aircraft, the vendor must supply evidence to a Federal Aviation Administration (FAA) representative that shows compliance to FAA's guidelines [11]. The certification process used in the aviation industry currently relies heavily on peer review and testing.

Many properties of a software product, such as correctness, or general as well as domain-specific safety, may be proven via deduction, synthesis, or other techniques [10, 1, 9, 5]. If the vendor proves the property and presents the proof to the FAA representative, the representative may check the proof and conclude that the system is indeed safe or correct relative to a specification. Such a scheme is known as proof-carrying code [10]. In a general setting, the vendor may not have to provide a proof, but some intermediate, semantic-based objects (collectively called a certificate) that help to establish the proof. The generalized category of approaches is known as semantic-based software certification.

The soundness of such a framework, however, is based on the assumption that programs constituting the "trust base" are correctly implemented.

Semantic based software certification was originally designed for the safe distribution of software in an untrusted environment. The approach can be adapted to software certification in avionics and other industries. A significant gap between research and industrial practice is the lack of qualified tools.[1].

It is necessary to distinguish two sets of tools. Besides the trust base, there are often other tools involved in a semantic-based software certification/reverification process. Naturally, the tools in the trust base should be more strictly scrutinized because their failure can allow errors to propagate to final products. It is expected that tools in the trust base will incur higher cost during qualification because of their higher criticality.

This suggests the need for architectural principles for designing tools to achieve desired trust goals. Choosing an optimal partitioning of components into trusted and untrusted sets becomes an important decision. Considering the high cost of qualification, the functionality needs to be decomposed in a way such that the combined cost of qualifying the tools is minimal.

Thus, the problem of selecting tools to qualify is a choice among approaches that have the required expressive power and trust attributes, and also allow a decomposition of the functionality that incurs acceptable qualification cost. Of particular interest in this paper is the case of abstraction-carrying code[14], which certifies temporal properties. Its trust base contains a model checker, which is an additional component beyond those of most other certification methods.

2 Abstraction-Carrying Code

In a sense, the concept of semantic-based program certification can be understood as decomposed program verification. Consider a vacuous program certification technique, where the certificate contains nothing. In this case, the regulatory authority (represented by and referred to hereafter as a DER, Designated Engineering Representative) has to verify the program on her own. If hints, for example, a loop invariant, are provided by the vendor, the DER is relieved of discovering this fact. But she needs to reverify that the loop invariant is indeed a loop invariant. On the trust base side, a data-flow analyzer that she may trust is now replaced by a simpler data-flow fact verifier. Because simpler programs are less expensive to qualify, and because we have assumed that a tool in the trust base requires a stricter, more expensive qualification process, the setting in which the vendor provides such a hint is beneficial in terms of tool qualification cost. As a principle, we should exploit this trade-off between the amount of information (size of certificate) provided by the vendor and the complexity of the trust base.

Traditionally, semantic-based program certification is proof-based. In theory, this scheme works for any properties that can be formalized in the underlying

[1] Other issues include, and are not limited to, recognition, training, and expressive power/tool support.

logic. And in practice, proofs can be generated for many safety properties even if approaches other than theorem proving are used. However, sometimes for general temporal properties, generating a proof may not be feasible. A different paradigm based on abstraction-carrying code is proposed. Table 2 lists several different, real or imaginary certification settings with their expressive power and associated trust base.

	Certification Method	Properties	Trust Base
1	Null Certificate	provable properties	every tool needed for proving
2	Touchstone	type and memory safety	VCGen and proof checker
3	AutoBayes	domain specific safety and memory safety	VCGen and proof checker
4	Any proof	provable properties	VCGen and proof checker
5	Abstraction-carrying code	temporal properties	VCGen, proof checker and model checker

The first row in the table refers to the no-certificate situation. The second row roughly corresponds to the setting of the original PCC work by Necula and Lee [10], where type safety and memory safety is of concern, where the trust base contains a verification condition generator (VCGen) and a proof checker. Row 3 represents the application of PCC techniques in the verification of domain-specific properties. For example, work by Denney et. al., automatically generates programs with proof-carrying code style proofs for domain specific safety properties[5]. Row 4 corresponds to the setting where the client uses a proof assistant (and probably a lot of human effort) to prove properties of concern.

Our focus is on Row 5, which corresponds to the abstraction-carrying code research by Xia and Hook [15, 14]. The idea is to apply predicate abstraction [6] and model checking to a program to verify an LTL property. Predicate abstraction may be automated by adopting counter-example driven predicate discovery [2, 4]. In this process, a predicate abstraction of the program is generated and passed to a DER. A DER will first verify that the abstract model is faithful to the program and then verify that the property does hold on the abstract model. The faithfulness check can be implemented much the same way as in PCC, that is, via a VCGen and a proof checker.

The proof-carrying code literature has elaborated how the VCGen and proof checker may be constructed in a simple manner. For example, a typed assembly language approach [8] can adopted for VCGen construction and a higher order logic framework [10] is used in proof checking. Our research is focused on how to restructure a model checker to achieve similar results.

3 Qualifiable Model Checkers

One of the initial design goals of abstraction-carrying code is to reduce the size of the certificate because of the need to transport it and check it at run time. In

certifying for administrative approval, however, size is not an important factor. There are approaches that generate proofs for certain sub-categories of properties after predicate abstraction/model checking[9, 7]. But such an approach may not be feasible for general LTL formulas. Therefore, the ability of abstraction-carrying code to certify temporal properties is still useful. In ACC, the trust base includes a model checker, which is absent from PCC. We are going to explore the trade-off between certificate size and complexity of the trust base to build a more cost-effectively qualifiable model checker.

The model checker to be used by a DER in abstraction-carrying code is different from a general purpose model checker: it checks an abstract model known as a Boolean program (BP)[3]. A typical BP statement tests if a propositional formula holds given an environment, represented as another propositional formula, and changes the state accordingly. Compared to a model checker for a target program, for example, the Java Pathfinder [13], the model checker for a Boolean program does not need to handle the semantics of the object language. In contrast, more than half of the code in Java Pathfinder implements the semantics of a virtual machine.

Still, this model checker is a fairly complicated program. For example, Moped [12], contains 10 K lines of C code, not counting the supporting BDD library. To reduce the size of this model checker while still achieving the requirements of re-verification, we resort to the tools that already exist in the trust base: the VCGen and the proof checker. Specifically, we use a hybrid approach to reuse some of the model checking work that would be performed by the vendor during the original analysis. This tool can be more complicated because it is not in the trust base. We enhance the vendor's model checker to record every transition made by the model checker. That is, for a transition (a BP statement t) that moves the system state (represented as a propositional formula) from s_1 to s_2, we note down the triple (s_1, t, s_2). Then the reduced model checker does not have to compute s_2, but just verify that $t(s_1)$ is s_2. Further, because the application of t to a state can be reduced to the test of satisfiability in propositional logic, we may simply keep a record of the piece of evidence that a proposition can be satisfied. This way, we can replace the SAT solver, or the BDD package used in the model checker with a Boolean evaluator. The DER will run this reduced model checker, which, when a satisfiability problem in the Boolean domain is needed, will just verify the proof presented by the vendor. In this way, the semantic engine needed to analyze the Boolean program is simplified.

We are at the very early stage of this investigation. We are aware that the complexity of the trust base is only one of the many factors involved in tool qualification. Still, we expect to build a prototype system that can be plugged into our previous implementation of an ACC system called ACCEPT/C and evaluate the effectiveness of this reduced model checker. Our effort also involves extending ACCEPT/C to support more interesting properties (that is, safety properties commonly seen as part of requirements in aviation systems). This part we have been initially successful [16]. Together, the simplified model checker and enhanced ACCEPT/C will allow exploration of the primary trade-off: deliver-

ing more detailed evidence at certification time in exchange for the benefits of reduced-complexity verification tools.

References

1. A. Appel. Foundational Proof-carrying Code. In *Proceeding of 16th IEEE Symposium on Logics in Computer Science (LICS)*, June 2001.
2. T. Ball. Formalizing counter-example driven predicate refinement with weakest preconditions. Technical Report MSR-TR-2004-134, Microsoft Research, 2004.
3. T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN2001, Lecture Notes in Computer Science 2057*, pages 103–122. Springer-Verlag, May 2001.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the Conference on Computer Aided Verification (CAV)*, pages 154–169, 2000.
5. E. Denney and B. Fischer. Certifiable program generation. In *Proceedings of Generative Programming and Component Engineering*, 2005.
6. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of Conference on Computer Aided Verification (CAV) 97, Lecture Notes in Computer Science 1254*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
7. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-Safety Proofs for Systems Code. In *Proceedings of Conference on Computer-Aided Verification (CAV)*, pages 526–538, 2002.
8. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
9. K. S. Namjoshi. Certifying Model Checkers. In *Proceedings of 13th Conference on Computer Aided Verification (CAV)*, 2001.
10. G. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
11. RTCA SC-167 and EUROCAE WG-12. Software considerations in airborne systems and equipment certification, December 1992.
12. S. Schwoon. Moped software. Available at http://wwwbrauer.informatik.tu-muenchen.de/~ schwoon/moped/.
13. W. Visser, S. Park, and J. Penix. Applying Predicate Abstraction to Model Check Object-oriented Programs. In *Proceedings of the 33rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*.
14. S. Xia. *Abstraction-based Certification of Temporal Properties of Software Modules*. PhD thesis, OGI School of Science and Engineering, Oregon Health and Science University, 2004.
15. S. Xia and J. Hook. Certifying Temporal Properties for C programs. In *Proceedings of Verification, Model Checking and Abstract Interpretation (VMCAI) 2004, Lecture Notes in Computer Science 2974*, 2004.
16. S. Xia, B. D. Vito, and C. Munoz. Predicate abstraction of engineering programs. Manuscript, 2005.

Reusing Proofs when Program Verification Systems are Modified

Bernhard Beckert, Thorsten Bormer, and Vladimir Klebanov
Institute for Computer Science
University of Koblenz-Landau
www.key-project.org

Abstract

In this position paper, we describe ongoing work on reusing deductive proofs for program correctness when the verification system itself is modified (including its logic, its calculus, and its proof construction mechanism).

We build upon a method for reusing proofs when the program to be verified is changed, which has been implemented within the KeY program verification system and is successfully applied to reuse correctness proofs for Java programs.

1. Motivation

Proof reuse in program verification is mostly thought of as a means to more easily construct a proof for the correctness of some program p in cases where a proof for a similar program p' (or the same program p with a slightly different specification) is already available.

If proofs are used as certificates for the correctness of programs, however, there is an even more important reason for reuse. One has to be able to reuse proofs in case the proof *system* is modified. While changing the program that is to be verified has only local effects in that only the proofs for that particular program are invalidated, modifying the verification system globally affects and potentially invalidates *all* proofs done so far. Since proofs that serve as certificates for program correctness need to be maintained over a longer period, possibly over many years, modifications to the proof system are to be expected over the lifetime of proofs. Thus, being able to reuse proofs when the system is modified is an indispensable feature of any program verification infrastructure that is put to serious practical use.

In [5], we have presented a method for reusing correctness proofs when the program changes. That method has been successfully implemented within the KeY system [7, 1] (see Sect. 2) to reuse correctness proofs for Java programs. It can handle many different types of changes in the program to be verified, such as adding/changing/deleting statements, changing (sub-)expressions, changing the control structure (e.g., by adding an if-statement), changing the class hierarchy, and overwriting inherited method implementations. It works well in practical everyday use; and only rarely are old proof attempts reused in a less than optimal way.

In this position paper, we describe ongoing work on the extension of our method to handle modifications of the verification environment instead of the programs to be verified. Possible modifications we consider include changes to the program logic used in the system, changes to the rules of the verification calculus, changes to the language used to represent these rules, and changes to the deductive engine of the proof system.

Note that we are not formalizing our method in a metalogic, as it's not helpful to achieve a working solution. Our subject is a particular kind of proof search procedure, and only valid proof objects can be constructed in any given prover version anyway (this also applies to loading a proof from a file). In a sense, the problem we are looking at is not one of logics.

2. Background

The KeY Project. The KeY system [7, 1] is a comprehensive environment for integrated deductive software design. Software developed with KeY can be formally proven correct, i.e., behaving up to the given specification. In the KeY process, the correctness of programs is formally proven by establishing the validity of Java Dynamic Logic formulas generated from the specification and the implementation of a program. This correctness is asserted by an explicit proof object.

The system is built on top of the CASE tool Borland Together ControlCenter, which is an enterprise-grade platform for UML-based software development. A version integrated with the popular open IDE Eclipse is also available. KeY augments this modeling foundation with an extension for formal specification, a verification middleware,

and a deduction component. Formal software specifications are written either in Object Constraint Language (OCL), which is part of the UML standard, or Java Modeling Language (JML). The KeY extension offers facilities for authoring, rendering and analysis of formal specifications. The verification middleware is the link between the modeling and the deduction component. It translates the model (the class diagram), the implementation (Java), and the specification (OCL/JML) into Java Dynamic Logic proof goals, which are passed to the deduction component. The verification middleware is also responsible for managing proofs during the development and verification process. The deduction component is a novel Java Dynamic Logic theorem prover that is used to actually construct proofs for the proof goal.

Java Dynamic Logic. Dynamic Logic (DL) can be seen to be an extension of Hoare logic (see [6] for an overview). It is a first-order modal logic with a modality $\langle p \rangle$ for every program p (we allow p to be any sequence of Java statements with the only restriction that they must not contain threads). In the semantics of these modalities a world w (called state in the DL framework) is accessible from the current world, if the program p terminates in w when started in the current world. The formula $\langle p \rangle \phi$ expresses that ϕ holds in *some* final state of p. Considering sequential Java programs, there is exactly one such final state for every initial state (if p terminates) or there is no final state (if p does not terminate). The formula $\phi \rightarrow \langle p \rangle \psi$ is valid if, for every state s satisfying precondition ϕ, a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds.

The KeY Calculus for Java Dynamic Logic. As usual for deductive program verification systems, we use a sequent-style calculus. The programs in Java DL formulas are basically executable Java code. The verification of a given program can be thought of as *symbolic code execution*.

The rules of the Java DL calculus [3, 1] operate on the first *active* command p of a program $\pi p \omega$; it is the focus of their application. The non-active prefix π consists of an arbitrary sequence of opening braces "{", labels, etc. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements like `throw` and `return` can be handled appropriately. The postfix ω denotes the "rest" of the program, i.e., everything except the prefix and the part of the program the rule operates on.

Since there is (at least) one rule schema in the Java DL calculus for each Java programming construct, we can here only give a simple but typical example, the rule schema for the `if` statement:

$$\frac{\Gamma,\ b = \text{TRUE} \vdash \langle \pi\ \ p\ \ \omega \rangle \phi \qquad \Gamma,\ b = \text{FALSE} \vdash \langle \pi\ \ q\ \ \omega \rangle \phi}{\Gamma \vdash \langle \pi\ \ \texttt{if(}b\texttt{)}\ \ p\ \ \texttt{else}\ \ q\ \ \omega \rangle \phi}$$

The rule has two premisses, which correspond to the two cases of the `if` statement. The semantics of this rule is that, if the two premisses hold in a state, then the conclusion is true in that state. In particular, if the two premisses are valid, then the conclusion is valid. Note, that this rule is only applicable if the condition b is known (syntactically) to be free of side-effects. Otherwise, if b is a complex expression, other rules have to be applied first to evaluate b.

The Taclet Mechanism. The KeY system provides a formalism for implementing rules (resp. rule schemata) called *taclets* [4]. As the name suggests taclets can be considered as lightweight, stand-alone tactics. They have a simple syntax and semantics and have means to represent explicitly (i) the pure logical content of a rule; (ii) restrictions or *guards* on the expected context and position of a rule application; (iii) heuristic information on whether and when a rule is applied automatically/interactively. Here is the same rule as above formulated as a taclet:

find $\langle \pi\ \ \texttt{if(}b\texttt{)}\ \ p\ \ \texttt{else}\ \ q\ \ \omega \rangle \phi$
replacewith $(\langle \pi\ \ p\ \ \omega \rangle \phi)$ *add* $(b = \text{TRUE} \vdash\ )$;
replacewith $(\langle \pi\ \ q\ \ \omega \rangle \phi)$ *add* $(b = \text{FALSE} \vdash\ )$

The KeY Proof Format. The KeY prover stores its proofs as a proof script consisting of a stream of rule names, application positions (as index into the sequent) and explicit schema variable instantiations if these cannot be inferred from the sequent. This format avoids excessive inclusion of formulas in the file, since these include programs and can be quite lengthy. On the other hand, this design does not perform gracefully if—for some reason—the currently constructed proof object does not match the form expected in the script.

3. Proof Reuse for Program Changes

In this section, we briefly describe our method presented in [5], which allows to reuse proofs when the program to be verified changes. It forms the basis for our work on reusing proofs when the verification system is modified.

The Need For Proof Reuse Upon Changes in Programs. Experience shows that the prevalent use case of program verification systems is not a single proof run. It is far more likely that a proof attempt fails, and that the program (and/or the specification, see Section 5.2) has to be revised. Then, after a small change, it is better to adapt and reuse the existing partial proof than to verify the program again from first principles. This is of particular advantage for deductive verification systems (which we consider here), where proof reuse reduces the number of required user interactions.

Features. The main features of our reuse method are:

(1) The units of reuse are single rule applications. That is, proofs are reused incrementally, one proof step at a time.

This allows to keep our method flexible, avoiding the need to build knowledge about particularities of the calculus, its rules, and the target programming language into the reuse mechanism.

(2) Proof steps can be adapted and reused even if the situation in the new proof is merely "similar" but not identical to the template.

(3) In case reuse has to stop because a changed part in the new program is reached that requires genuinely new proof steps, reuse can be resumed later on when an unaffected part is reached.

Basic Ideas. The rules of the calculus are represented by rule schemata (taclets). Thus, at each proof step, there are three choices that the reuse facility—like every incremental proof construction method—has to make: (a) the rule (schema) to be applied, (b) the goal/position where it is applied (which we call the "focus" of the rule application), and (c) instantiations for schema variables.

Our goal is to make—if possible—the same choices as in the template proof. But that requires us to generalize and extract the essence of the choices in the old proof such that it can be applied to the (similar but different) situation in the new proof.

For finding the rules that are candidates for choice (a), such a generalization is readily available. The rule *schemata* (i.e., the schematic representations of the rules) are natural generalizations of particular rule applications. They are defined by the developer of the verification calculus who has the required insight to know what the essence of a rule application is. We then adhere to the overall succession of rule schema applications in the template proof. But, since proofs are not linear, at each point in time there can still be several candidate rules that compete for being used first.

Choice (b), i.e., the point where a candidate rule is to be applied, is more difficult as it is hard to capture the essence of a formula or sequent. To solve this problem, we use a syntactical similarity measure on formulas. Fortunately, there is usually only a moderate number of possibilities, because program verification calculi are to a large degree "locally deterministic". That is, given a partial (new) proof, there is for most rule schemata only a small number of potential application foci.

Finally, to make choice (c), schema variable instantiations are computed by matching the rule schema against the chosen focus of application. Schema variables that do not get instantiated that way, e.g., quantifier instantiations, are simply copied verbatim from the old proof.

Finding Reusable Subproofs. Our main reuse algorithm requires an initial list of reuse candidates. These initial candidates, which are rule applications in the old proof, can be seen as the points where the old proof is cut into subproofs that are separately reusable. They are the points where reuse is re-started after program changes required the user or the automated proof search mechanism to perform new rule applications not present in the old proof. The choice of the right initial candidates is crucial for reuse performance.

The way initial candidates are computed depends on the way the program (and thus the initial proof goal) has changed. For changes affecting single statements (local changes) we extract the differences right from the source files, using the GNU diff utility. Non-local changes, such as renaming of classes or changes in the class hierarchy, cannot be detected in a meaningful way by the standard diff algorithm; the user has to announce the changes separately. We are also investigating application of the recently emerged techniques for difference detection in object-oriented programs [2].

Adaptability. Our reuse approach is very flexible. The only part that is to some extent adapted to the target calculus is the similarity measure on formulas. But even that does not incorporate any knowledge about particular rules but only some limited information about the target programming language (Java in our case) and general properties of the calculus (e.g., that rules are typically applied at the beginning of a program).

4. Proof Reuse for Modifications of the Verification System

4.1. Recertification Strategy Catalogue

In this section, we discuss different modification scenarios that we have encountered during the six years of development of the KeY system. All verification systems are subject to evolution, and the ones that persistently store proof-relevant information (proof scripts, lemmas, abstractions, program invariants, etc.) have to deal with similar problems as we did. We believe that developers of other deductive verification systems can profit from our experiences.

We classify the recertification strategies that are possible responses to modifications of the verification environment as follows:

A No action necessary. The old proof can be loaded without modification.

B Automated recertification with the help of additional information that has been provided at time of change.

C Machine-supported recertification while necessary additional information is inferred during the process. In this case we assume that the old proof can be loaded into the system with the corresponding set of rules.

D Same as C, but in case the old proof cannot be loaded into the system. Then, just the information available in the stored proof file is available.

E No action possible/intended.

$$\frac{\begin{array}{c}\Gamma,\ a = null\ \vdash\ \langle\pi\ \texttt{NPE;}\ \omega\rangle\phi \\ \Gamma,\ a \neq null \wedge (i < 0 \vee i \geq a.length)\ \vdash\ \langle\pi\ \texttt{AOBE;}\ \omega\rangle\phi \\ \Gamma,\ a \neq null \wedge i \geq 0 \wedge i < a.length\ \vdash\ \{a[i] := val\}\langle\pi\ \omega\rangle\phi\end{array}}{\Gamma\ \vdash\ \langle\pi\ \ a\texttt{[}i\texttt{]=}val\ \ \omega\rangle\phi}$$

$$\frac{\begin{array}{c}\Gamma,\ a = null\ \vdash\ \langle\pi\ \texttt{NPE;}\ \omega\rangle\phi \\ \Gamma,\ a \neq null \wedge (i < 0 \vee i \geq a.length)\ \vdash\ \langle\pi\ \texttt{AOBE;}\ \omega\rangle\phi \\ \boxed{\Gamma,\ a \neq null \wedge i \geq 0 \wedge i < a.length \wedge \neg storable(val, a)\ \vdash\ \langle\pi\ \texttt{ASE;}\ \omega\rangle\phi} \\ \Gamma,\ a \neq null \wedge i \geq 0 \wedge i < a.length \boxed{\wedge storable(val, a)}\ \vdash\ \{a[i] := val\}\langle\pi\ \omega\rangle\phi\end{array}}{\Gamma\ \vdash\ \langle\pi\ \ a\texttt{[}i\texttt{]=}val\ \ \omega\rangle\phi}$$

Figure 1. A rule for array assignment: initial and revised version. Differences are boxed.

In the following discussion of different types of modifications, we indicate what we believe is the right kind of strategy to handle the respective modification. We currently work on developing methods for strategies B, C, and D.

4.2. Changes of the Logic Syntax

The presence of a rich program and logic vocabulary within the same formula makes designing a usable and at the same time parseable logic syntax quite a challenge. Several iterations were necessary to obtain a satisfactory solution.

The quantifier notation of `exists x:int.prop(x)` was changed in order to allow fully qualified sort names, as in `\exists java.lang.Object o; prop(o)`. The diamond modality notation had to be modified from the simple `<program>formula` to `\<program\>formula` in order to allow `a<b` in place of `lt(a,b)`. Proposed recertification strategy: B, as one could parse old proof versions with the associated old parser and then transform the abstract representation into the new format. Furthermore, stored KeY proofs rarely mentions formulas explicitly.

4.3. Changes of the Taclet Language

The taclet language used to define the rules of the KeY prover is also subject to change. As clashes between taclet declaration keywords and JAVA identifiers became apparent, an escaping mechanism was put in place (`find` $\rightsquigarrow$ `\find`). Altogether this kind of change is transparent in the stored proofs, as these only reference taclet names. Recertification strategy: A. The semantics of the taclet language has turned out to be exceedingly stable.

4.4. Changes in Parser/Disambiguation

Between the levels of syntax and semantics are changes in parsing and disambiguation of logical expressions. An example is a modification of the associativity of logic operators. The interpretation of the expression $A \wedge B \wedge C$ has changed from $(A \wedge (B \wedge C))$ to $((A \wedge B) \wedge C)$. In addition, the precedence between the state update operator and arithmetic operators were changed in favor of the update so that $\{update\}a + b$ evaluates to $(\{update\}a) + b$ instead of the former meaning $\{update\}(a + b)$. Recertification strategy: B. The old parser can be used to produce an AST, from which an equivalent linearization for the new parser can be generated using explicit brackets.

4.5. Changes in Formalization of the JAVA Language Semantics

Sometimes minor errors in the symbolic execution rules of the KeY calculus have to be fixed. This cannot be ruled out, since one can never arrive from an informal specification at a formal one by formal means. The KeY project on regular bases performs the only measure suitable to mitigate this: cross-checking our rules with other formalizations of JAVA. A recent check of this kind [8] has discovered a missing case in our array assignment rule. The erroneous rule and its correction are presented in Figure 1. Recertification strategy: C. Small local changes allow a similarity-guided proof reuse.

4.6. Changes in the Logical Structure of the Rules

At one point all rules containing a potential case distinction have been reformulated from the form (here's an example)

$$\frac{\Gamma\ \vdash\ ((a > b) \rightarrow \langle\pi\ \ l\ =\ \texttt{true;}\ \ \omega\rangle\phi) \wedge (\neg(a > b) \rightarrow \langle\pi\ \ l\ =\ \texttt{false;}\ \ \omega\rangle\phi)}{\Gamma\ \vdash\ \langle\pi\ \ l\ =\ a\ >\ b\ \ \omega\rangle\phi}$$

to a form employing a conditional formula

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{if}\,(a > b) \\ \langle \pi\ \ l\ =\ \texttt{true;}\ \ \omega\rangle\phi\ \mathit{else}\ \langle \pi\ \ l\ =\ \texttt{false;}\ \ \omega\rangle\phi\ \mathit{fi}\end{array}}{\Gamma \vdash \langle \pi\ \ l\ =\ a\ >\ b\ \ \omega\rangle\phi}$$

which has the advantage that one has to reason about the condition only once. Recertification strategy: C, same as above.

4.7. Changes in the Execution Engine

As noted in Section 2, it is important that the expected shape of the proof object implicit in the proof script matches the actual construction. This concordance can be disrupted if the execution engine of the prover is either not deterministic or is purposefully changed.

Ordering of Proof Branches/Formulas. One degree of freedom left by the calculus is the way (i.e., position of) formulas are added to the sequent, and the ordering of the newly generated subgoals whenever a rule has several premisses.

Recertification strategy: D, as it's not possible to load the old proof with a changed system.

The Link to the Program Model. The method call rule of the KeY calculus simulates dynamic binding by a case distinction over all possible classes that offer a suitable implementation of the called method. The ordering of branches is a potential nondeterminism source, which has been eliminated recently by applying alphabetical sorting. Recertification strategy in case of change: D.

Changes in Logic Data Structures. Since stored proofs contain numerical indices into the internal representation of logical entities (formulas, terms), changes in this data structure affect the loading of proofs. Except one such change, the representation has remained stable so far. Recertification strategy: D as this problem class is similar to the one dealing with the ordering of proof branches and formulas.

5. Further Related Issues

5.1. Changes of the JAVA Platform

In spite of Sun's policy of upward source compatability, new versions of the JAVA platform may bring changes to the semantics of existing programs. We want to mention here the introduction of new keywords and APIs (code has to be rewritten to avoid clashes), bugfixes and updates to the JAVA libraries (e.g., fixing the method `StringBuffer.append(StringBuffer)` to be thread-safe or method `BigInteger.isProbablePrime(int)` not to report false for certain primes), or revisions of the JAVA Memory Model (as proposed in JSR 133). The range of necessary recertification actions stretches from A to E, depending on the particular case (e.g., whether the specification or the implementation of library methods was used in proofs, etc.).

5.2. Changes in Program Specification

The complementary case of a change in the program is a changing specification. While the same techniques of proof reuse should be applicable to some extent, a specification is usually a higher-level description, and small changes are likely to lead to bigger disruptions in proofs. We will not pursue this issue here, as it affects individual problems only.

6. Conclusion

We have discussed possible modifications occurring during the development and evolution of software verification systems. A similarity-based method for proof reuse, which has already been successfully implemented in the KeY system, has been presented that handles program changes. Currently we extend and adapt this method to implement recertification strategies of types B, C, and D.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling (SoSysM)*, 4:32–54, 2005.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 2–13, Linz, Austria, September 2004.

[3] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[4] B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

[5] B. Beckert and V. Klebanov. Proof reuse for deductive program verification. In J. Cuellar and Z. Liu, editors, *Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China*. IEEE CS Press, 2004.

[6] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[7] KeY Project. Website at `www.key-project.org`.

[8] K. Trentelman. Proving correctness of JAVACARD DL taclets using Bali. In B. Aichernig and B. Beckert, editors, *Proceedings, 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE CS Press, 2005.

Software Certification Management
How Can Formal Methods Help?

Dieter Hutter
German Research Centre for Artificial Intelligence (DFKI GmbH),
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany,
e-mail: hutter@dfki.de

Abstract

The formal development of industrial-size software is an error-prone and therefore evolutionary process. We report on our efforts to implement an assistance tool that helps us to anticipate the effects of changes in formal specification, to retrieve existing specifications and adapt them to new situations, to determine the "minimal" sets of proof obligations that will newly arise or which proofs have to be re-tackled again, and to adjust the old proofs to the new conditions. As a result of this work we outline an underlying theoretical framework for a general repository to maintain mathematical or logic-based documents while keeping track of the various semantical dependencies between different parts of various types of documents (documentations, specifications, proofs, etc).

1 Introduction

In the last decade Formal Methods have been successfully applied to specify and verify security or safety critical systems. In the area of security, the formal specification (and partly also verification) of smartcards became a necessity to comply with the security requirements of their users. Car manufacturers start to use formal methods to get the more and more complex devices and sophisticated interaction between them under control. Formal software development paradigms are closely related to the waterfall model. Starting with a formal (textual) specification, it is translated into a logic based formalism, proof obligations are calculated to guarantee the security or safety properties, and finally these obligations have to be proven usually with the help of model checking or theorem proving. However in all applications so far, development steps turned out to be flawed and errors had to be corrected. The search for formally correct software and the corresponding proofs is more like a formal reflection on partial developments rather than just a way to assure and prove more or less evident facts. Figure 1 illustrates this typical process.

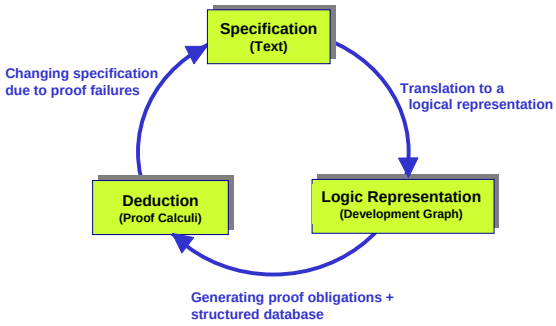


Figure 1: Formal Development Cycle

2 Development Graphs

The painful experience of this evolutionary character of applying formal methods resulted in the development of the MAYA system [2, 3] to maintain the formal software development process. We envisioned an assistance tool that helps us to anticipate the effects of changes in the specification, to retrieve old specifications and adapt them to new situations, to determine the "minimal" sets of proof obligations that will newly arise or which proofs have to be retackled again, and to adjust the old proofs to the new conditions.

MAYA was a first step towards such a system. It supports an evolutionary formal development since it allows users to specify and verify developments in a structured manner, incorporates a uniform mechanism for verification *in-the-large* to exploit the structure of the specification, and maintains the verification work already done when changing the specification. Maya relies on development graphs as a uniform representation of structured specifications, which enables the use of various (structured) specification languages like Casl [4], OMDoc and VSE-SL [6] to formalize software development. To this end Maya provides a generic interface to plug in additional parsers for the support of other specification languages. Moreover, Maya allows the integration of different theorem provers to deal with proof obligations arising from the specification, i.e. to perform verification *in-the-small*.

Textual specifications are translated into a structured logical representation called a *development graph* [1, 5], which is based on the notions of consequence relations and morphisms and makes arising proof obligations explicit. The user can tackle these proof obligations with the help of theorem provers connected to Maya like Isabelle [12] or Inka [8].

A failure to prove one of these obligations usually gives rise to modifications of the underlying specification. Maya supports this evolutionary development process as it calculates minimal changes to the logical representation readjusting it to a modified specification while preserving as much verification work as possible. If necessary it also adjusts the database of the interconnected theorem prover. Furthermore, Maya communicates to the attached provers explicit information how the axiomatization has changed and also retrieves former proofs of the same problem (that are now invalidated by the changes) to allow the theorem provers to reuse them. In turn, information provided by the theorem provers about the computed proof is used to optimize the maintenance of proofs during the evolutionary development process.

3 Transformational Development

While MAYA supports the adaption of formal developments to changed specifications, it primarily focuses on the computation of differences between old and new specification and to maintain and propagate these changes along the development cycle illustrated in Figure 1. However, due to the complexity of this process there is no guarantee that "simple" changes in the specification will be adequately supported by the system when it comes to the adaptation of proofs. Starting with [14] we worked also on defining building blocks to transform developments as a whole. The idea is to incorporate the knowledge about the way a specification is changed into rules how to adapt the existing proofs simultaneously. This results in a set of basic transformations operating on developments and changing specifications *together* with their proof obligations and proofs in parallel and returning an adapted new development. Typical examples are the change of abstract datatypes by adding or removing constructors, the change of parameters of function and predicates, and the change of axioms by adding or removing conditions [13]. Using these basic transformations allows a developer to predict the effects of his changes to the entire development since each of these basic transformations will change specification, proofs and proof obligations in a predetermined and controlled fashion.

4 A More General Approach

Developing the MAYA system, which is specialized to the maintenance of formal developments based on

algebraic specifications, we realized that the underlying methodology is rather general and independent of the used logical representation of specifications and proofs but relies heavily on the structure of dependencies between objects and properties and how these dependencies can be decomposed along a given structure.

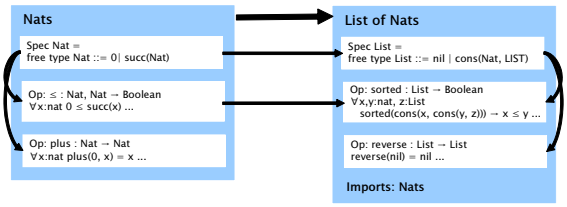


Figure 2: Semantic Dependencies in Specifications

In formal developments the semantics of (structured) objects depends on the semantics of their sub-objects used for their definition (axiomatic dependencies). While there is no way to scrutinize changes of axiomatic dependencies in case of intentional (or purposeful) changes of the development, there is a need to inspect axiomatic dependencies in case of mechanical changes as they occur, for instance, during the (automatic) merge of two branches of a development.

Properties between (structured) objects can be postulated and proven within a development. Similar to the objects under consideration, the proofs of properties about such objects are also structured. Hence, such a proof depends on (or decomposes into) properties of sub-objects which gives rise to *deduced dependencies* between different properties. Changing the development may render proofs invalid since either some basic property does no longer hold or the way the problem was decomposed is no longer appropriate.

The ability to decompose properties along the structure of the concerned objects allows us to localize the effects of changes. A property between structured objects (theories) is decomposed (according to some decomposition rules) to properties between their sub-objects (local axioms). Typically, these properties between structured objects have to be independent of the environment in which these objects might occur. As long as the concerned structured objects are unchanged any change of the overall development will not inflict the already proven or postulated properties between these objects.

5 Repository Supporting Distributed Development

As a consequence we now work on a repository [7] to maintain all sorts of dependencies between various parts of a formal development or even informal documents [9]. The main goal of such a logic-based repository is to ease the development of mathematical or logic based knowledge consisting of entities such as axioms, definitions, theorems, proofs and informal documentations (sometimes including semantic annotations). As the development of a software project or of mathematical knowledge is distributed, also the repository has to support distributed developments. We propose a CVS-like infrastructure to determine the differences between two versions, to calculate the necessary changes to update a local repository to the current state, and to integrate two rival developments into a merged variant. However while text-lines might be appropriate to structure pure text documents, this approach fails completely in logic-based documents. A single text line may contain independent terms or a single term could be spread over many text lines. Undiscovered "semantic" conflicts may occur if two users change different text lines that are both part of the description of a single term. Changing the arity of a signature symbol in a document typically requires to change its arity in all the occurrences of the symbol. Therefore, we use the more semantically adequate representation of acyclic directed graphs as the general structure underlying the documents under consideration and redefine the CVS notions of diff, patch and merge in this context.

In a second phase we add semantical dependencies between different parts of a document to detect semantical conflicts, for instance, when merging different versions of a document that result from chang-

ing different but semantically still dependent parts of a document by different users. When merging documents, semantic conflicts occur if semantically related documents are changed independently by two users. Decomposing dependencies along the structure of the objects allows one again to narrow down potential semantic conflicts: conflicts of composed objects only arise if there is a conflict between dependent sub-objects.

6 Conclusion

Inspecting the ideas of MAYA we discovered that most of the work related to the management of change does not require a deep knowledge of the semantics of the underlying specification languages. Instead the management of change solely operates on the structure of the objects under consideration and on how proposed properties can be decomposed to properties of their sub-objects.

The ultimate goal is to support generic structuring mechanisms as they occur in various domains by developing a system supporting these mechanisms while outsourcing application specific parts into modules attachable to the system. This would allow us to instantiate such a system for various purposes, like for instance in formal methods (cf. MAYA [2]), program development, or even maintaining informal documents like, for instance, course materials (cf. MMISS [9]).

References

[1] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. In *Recent Developments in Algebraic Development Techniques, WADT'99, Bonas, France*, Springer LNCS 1827, 2000.

[2] S. Autexier, D. Hutter, T. Mossakowski and A. Schairer. The Development Graph Manager MAYA. In *Proceedings 9th International Conference on Algebraic Methodology And Software Technology, AMAST2002*. Springer, LNCS 2422, 2002

[3] S. Autexier and D. Hutter. Mind the Gap - Maintaining Formal Developments in MAYA, In *Mechanising Mathematical Reasoning, Essays in honor of J.H. Siekmann*, Springer-Verlag, LNCS 2605, 2005

[4] B. Krieg-Brückner and P. Mosses (eds). CASL Reference Manual, Springer, LNCS 2960, 2004

[5] D. Hutter. Management of Change in Verification Systems. In *Proceedings 15th IEEE International Conference on Automated Software Engineering, ASE-2000*, IEEE Computer Society, 2000.

[6] D. Hutter et al. Verification Support Environment (VSE), *Journal of High Integrity Systems*, Vol. 1, pages 523–530, 1996.

[7] D. Hutter. Towards a Generic Management of Change. In *Workshop on Computer-Supported Mathematical Theory Development, International Joint Conference on Automated Reasoning'04*, Cork, Ireland, 2004

[8] S. Autexier, D. Hutter, H. Mantel, A. Schairer: System Description: INKA 5.0 - A Logic Voyager. In H. Ganzinger, *CADE-16*, Springer, LNAI 1632, 1999.

[9] B. Krieg-Brückner, D. Hutter, C. Lüth, E. Melis, A. Pötsch-Heffter, M. Roggenbach, J. Smaus and M. Wirsing. Towards MultiMedia Instruction in Safe and Secure Systems. In: *Recent Trends in Algebraic Development Techniques, (WADT-02)*. Springer, LNCS 2755, 2003

[10] T. Mossakowski and P. Hoffman and S. Autexier and D. Hutter. Part IV: CASL Logic. In: [4], 2004

[11] T. Mossakowski, S. Autexier, and D. Hutter. Development Graphs – Proof Management for Structured Specifications. *Journal of Logic and Algebraic Programming*, Special Issue on Algebraic Specification and Development Techniques, Elsevier, 2005 (forthcoming)

[12] L.C. Paulson. *Isabelle - A Generic Theorem Prover*, Springer LNCS 828, 1994.

[13] A. Schairer. Transformations of Specifications and Proofs to Support an Evolutionary Formal Software Development. PhD thesis (submitted), Saarland University, 2005

[14] A. Schairer and D. Hutter Proof Transformations for Evolutionary Formal Software Development. In: *Proceedings of the 9th International Conference on Algebraic Methodology And Software Technology, AMAST-2002*, LNCS 2422, 2002

Application of a Commercial Assurance Case Tool to Support Software Certification Services.

Luke Emmet <loe@adelard.com>
Sofia Guerra <aslg@adelard.com>

Adelard, Drysdale Building,
Northampton Square, London EC1V 0HB, UK

1 Introduction

Many industry sectors require a documented case that the system will meet its critical requirements; this documented case is often called an "assurance case". In the past, safety justifications tended to be *implicit* and *standards-based*—compliance to accepted practice was deemed to imply adequate safety. This approach works well in stable environments where best practice is supported by extensive experience, but in those sectors adopting fast moving technologies, a more explicit goal-based approach has been advocated, which can accommodate change and alternative strategies to achieve the same goal.

Goal-based approaches have been increasingly adopted in a range of industry sectors, where claims (or goals) are made about the system and arguments and evidence is presented to support that those goals are met. Goal-based approaches are more flexible as they focus directly on the critical requirements and they are more attuned to the ways in which sophisticated engineering arguments are actually made.

Our contention is that developing a software certificate is akin to the process used for system assurance argument. In fact, if software certification is taken as the "demonstration of the reliability, safety, or security of software systems in such a way that it can be checked by an independent authority" [7], there is no obvious clear distinction between software certification and the development of a safety case in a regulated sector (although for non-regulated sectors, there might not be the review by an independent authority).

Based on conceptual work by Toulmin [1], recent assurance and safety case notations have been developed, including Claims-Argument-Evidence [2], [3] and Goal Structuring Notation [6].

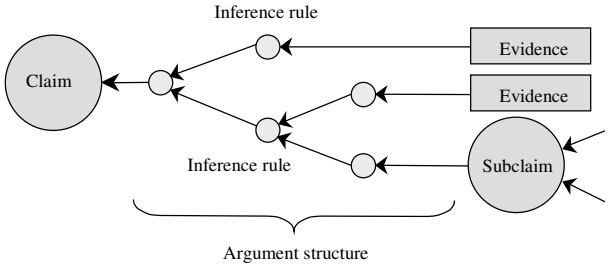


Figure 1: Generic argument structure

The components of the assurance cases are similar to those of a software certification process. They are:

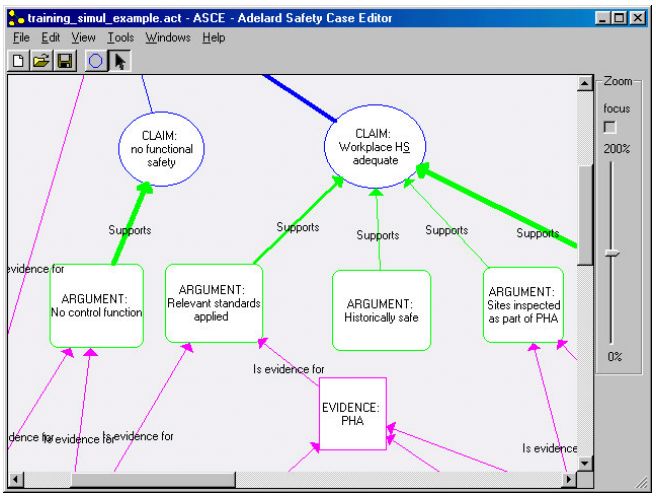
- *Claims* about a property of the system or sub-system, such as functional properties, RAM (Reliability, Availability, Maintainability) properties, security or usability.
- *Argumentation* nodes that describe the approach to satisfying the claim and provide the rationale for how the collected evidence is to be used to support the claims being made about it. We distinguish between deterministic or analytical (e.g. formal proof, exhaustive testing), probabilistic (e.g. MTTF, reliability testing) and qualitative (e.g. compliance with QMS or with safety standards).

- Process and product *evidence* for the software components, including design documentation quality system documentation, testing evidence, COTS product evaluations, etc.

In Adelard we have been addressing the justification of software-based systems for 18 years. This work has included research and development into the nature, structure and content of the justification of software. The concepts we have developed have been especially used for justifying safety of software-based systems, but these same concepts have been used for reasoning about other system properties such as dependability, reliability and security [8].

2 The ASCE system

ASCE (The Assurance and Safety Case Environment) is a flexible graphical hypertext system used for the development, review and maintenance of assurance and safety cases and other structured technical documentation [4], [5]. It is a fully supported commercial[1] tool currently being used by a wide range of organisations for the development, and assurance of safety-related systems.



Its main technical aspects are as follows:

- Flexible and intuitive *graphical editor* supporting a range of notations, such as Claims-Arguments Evidence and GSN. Other notations can be supported through the use of domain-specific "schemas".
- Each node contains a *structured HTML* narrative field supporting embedded fine grained cross-references to other elements in the network, shared files and Internet resources.
- User *extensible plugins* to support specific applications using ASCE, and integration with other technologies, including
    - *Validated content services*—so-called Dynamic Narrative Regions (DNRs). This allows narrative to be generated from external sources and integrated into an ASCE node narrative. ASCE checksums all DNR content and enables such external dependencies to be validated to see if they have changed since last imported.
    - *Structural and content analysis services*. For example algorithms can be developed that identify all the evidence nodes that provide support for a selected claim. An impact analysis algorithm could identify all claims reliant on a particular piece of evidence.
    - *Powerful reporting capabilities*. Custom reports can be defined by using plugins, which augment the standard capabilities of exporting to a collection of HTML files or to a linear word processed document.
- Its *XML data structure* allows external analysis of ASCE files, and supports long-term use of the data within an ASCE network.

[1] ASCE is free for non commercial teaching and academic research purposes.

- Includes an *ASCE difference tool*, allowing the analyses of two versions of an ASCE file for detecting structural and narrative differences between them. The ASCE difference tool can be used to support review or audit activities.

3 Software Certification

The ASCE system as it is currently designed can be readily configured to support software certification processes. Examples of scenarios of use are described in the following subsections.

3.1 Development of software certification case structure

The overall certification argument is defined in terms of claims being made for the system and its component subsystems. ASCE supports the development and clear presentation of the strategy used for justifying the software certification, showing how these claims will be met (and are progressively met as part of the certification process) and the evidence that has been identified to support the claims. The argument presented will ultimately be audited or reviewed (see Section 3.3 below).

3.2 Evidence integration

As the certification process evolves, the argumentation strategies may be developed and the evidence that supports the claims will become available. In some cases, the argumentation strategy may need to be modified if the evidence is stronger or weaker than originally envisaged.

One particular kind of evidence may be a certification case for subsystem elements. These can be integrated as evidence nodes in the overall certification case using DNR elements in the node narratives. This supports the notion of certificate hierarchies.

Overall, the diverse supporting evidence will be integrated using a number of methods:

- Narrative entered directly into the ASCE file.
- Hyperlinks to external files containing the evidence.
- DNR elements for critical evidence (e.g. testing evidence may need to be integrated, showing the actual status of the test results). This provides the strongest level of validation to external information sources.

A particular DNR might be developed that allows the certification case to particularly refer to the full configuration of a software item. This might require a simple integration with the configuration management tool that is being used.

3.3 Auditing and review

The certification case can be audited by reviewing the contents of the ASCE file. Given that external evidence is directly linked in, it is possible for the auditor to graphically view, and ultimately follow chains of dependencies from the claims being made down to the underlying evidence. The ASCE difference tool might be used to review differences since the previous version. The ASCE tool also supports clear notation for making which nodes have been audited and completed or accepted. The auditor can easily keep track of what aspects of the arguments have been accepted by looking at the nodes' annotations of the top-level graphical argument.

3.4 Certificate publication and revocation

Although the certification case will be considered the formal controlled information artefact, it may be required (e.g. for contractual purposes) to publish a paper document as a formal deliverable. In this case, the standard reporting tools would be used to create such a publication.

Before publication, the software certificate manager would validate all the DNR elements in the case. This would determine whether the underlying evidence had changed since it was last checked.

- *Unchanged* - if the certification case has been reviewed and audited, the certificate is still valid and can be published against the particular build.
- *Changed* - the underlying evidence has changed. Therefore the certification case must be reviewed to determine whether the top-level claims are still valid. The ASCE impact analysis algorithms could be used to help the reviewer determine which claims depend on the evidence changed and might need to be revisited.

4 Future directions

Given the extensible architecture of the tool, there are a number of aspects of use that can be supported with additional notations and plugins. These include:

- **Formal modelling and reasoning engines** – we have already implemented notations and algorithms (e.g. to support Fault trees) that formally analyse the structure of the representation used and propagate information across it. We are actively investigating integration with additional reasoning engines, although we anticipate there may be difficulty in faithfully in implementing the full underlying semantics of the reasoning engine.

- **Notation enhancements** – each core notation schemas in ASCE has been tuned for a particular purpose. One of our active areas of development is to enhance existing notations and develop new ones to support particular forms of analysis, and to distinguish particular kinds of evidence used (e.g. analytical, probabilistic and qualitative). Depending on the schema definition it is possible for different display rules to visually amplify this information and present it as decoration on the main display (e.g. as “traffic lights”).

- **Modular cases and integration** – as larger arguments are developed it is necessary to refactor the argument and modularise it to allow for different elements to be maintained by different stakeholders and to evolve separately. Existing facilities such as DNRs allow for changes to be managed across the interfaces between such modules. However, more work is needed in the future to explore the issues arising from such use in a way that supports pragmatic usage but at the same time provides a useful degree of automated integration as the collection of interdependent modules evolve.

5 Conclusions

Assurance and safety cases are mature concepts that already have wide use to support the assurance of dependable systems. Such concepts can be applied to software certification, in that claims need to be established and backed up by evidence about the product or its development process.

There are a number of open issues that are relevant for both assurance cases and software certification. In [8] we summarised some of the research areas we have been working, which included the use of formality and models to support the validation of the assurance case, the relationship to standards and how the safety case goal-based approach can be deployed in other application areas.

ASCE is a commercial tool used to develop and manage assurance and safety cases, and in this paper we have outlined how it could be used to support the development and management of software certificates. Together with the ASCE user community, we are actively developing the product and its associated notations and plugins. Our aim is to support an increasingly wide range

of application scenarios associated with heterogeneous information management and the demonstration of systems assurance.

6 References

[1] Toulmin, S.E. (1958) The Uses of Argument, Cambridge University Press, Cambridge, England.

[2] Bishop, P. & Bloomfield, R. A Methodology for Safety Case Development, Safety-Critical Systems Symposium, Birmingham, UK, Feb 1998

[3] Adelard (1998) ASCAD—The Adelard Safety Case Development Manual ISBN 0 9533771 0 5

[4] ASCE home page: http://www.adelard.com/software/asce

[5] Luke Emmet & George Cleland, Graphical Notations, Narratives and Persuasion: a Pliant Systems Approach to Hypertext Tool Design, in Proceedings of ACM Hypertext 2002 (HT'02), June 11-15, 2002, College Park, Maryland, USA.

[6] Kelly, T. Arguing Safety A Systematic Approach to Managing Safety Cases (1998). PhD Thesis, available at http://www.cs.york.ac.uk/ftpdir/reports/YCST-99-05.ps.gz

[7] Call for Papers - ASE Workshop on Software Certificate Management (SoftCeMent)

[8] P.G. Bishop, Robin Bloomfield and Sofia Guerra. The future of goal-based assurance cases. In Proceedings of Workshop on Assurance Cases. Supplemental Volume of the 2004 International Conference on Dependable Systems and Networks, pp. 390-395, Florence, Italy, June 2004.

Author Index

Notes